BEST AVAILABLE IMAGES

Defective images within this document are accurate representation of The original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

THIS PAGE BLANK (USPTO)

PA 309513

THE UNITED STATES OF AMERICA

TO ALL TO WHOM THESE PRESENTS SHALL COME:

UNITED STATES DEPARTMENT OF COMMERCE

IL 00/00540

United States Patent and Trademark Office

4

September 29, 2000

THIS IS TO CERTIFY THAT ANNEXED HERETO IS A TRUE COPY FROM THE RECORDS OF THE UNITED STATES PATENT AND TRADEMARK OFFICE OF THOSE PAPERS OF THE BELOW IDENTIFIED PATENT APPLICATION THAT MET THE REQUIREMENTS TO BE GRANTED A FILING DATE UNDER 35 USC 111.

APPLICATION NUMBER: ***60/152,849***
FILING DATE: ***September 08, 1999***

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

By Authority of the
COMMISSIONER OF PATENTS AND TRADEMARKS

L. Edelen

L. EDELEN
Certifying Officer

A1PRGN

09/08/99 U.S. PTO 60/152849

09/08/99 U.S. PTO

PROVISIONAL APPLICATION FOR PATENT COVER SHEET

This is a request for filing a PROVISIONAL APPLICATION FOR PATENT under 37 CFR 1.53 (b)(2).

Docket Number	204,238	Type a plus sign (+) inside this box →	+

INVENTOR(s)/APPLICANT(s)

LAST NAME	FIRST NAME	MIDDLE NAME/ INITIAL	RESIDENCE (CITY AND EITHER STATE OR FOREIGN COUNTRY)
KAGAN	Michael		71 Hashomer Street Zichron Yaakov 30900 Israel

TITLE OF THE INVENTION (280 characters max)

ADVANCED INPUT/OUTPUT DEVICE WITH PCI PORT

CORRESPONDENCE ADDRESS (including country if not United States)

ABELMAN FRAYNE & SCHWAB
Attorneys at Law
150 East 42nd Street
New York, NY 10017

ENCLOSED APPLICATION PARTS (check all that apply)

X	Specification	Number of Pages: 120	☐	Small Entity Statement	
☐	Drawing(s)	Number of Sheets:	☐	Other (specify):	

METHOD OF PAYMENT (check one)

X	A check or money order is enclosed to cover the Provisional filing fees	PROVISIONAL FILING FEE AMOUNT ($)	$ 150.00
X	The Commissioner is hereby authorized to charge any additional filing fees required and credit Deposit Account Number: 01-0035		

The invention was made by an agency of the United States Government or under a contract with an agency of the United States Government.

☒ No.

☐ Yes, the name of the U.S. Government agency and the Government contract number are: _____

Respectfully submitted,

SIGNATURE _____ Date 09/08/99

TYPED or PRINTED NAME Stewart J. Fried REGISTRATION NO. (if appropriate) 20,694

☐ Additional inventors are being named on separately numbered sheets attached hereto

MT101 architecture specification

Rev0.3

Michael Kagan
Mellanox Technologies

Table of contents

List of figures

List of Tables

System Architecture Overview

System block diagram

MT101 is an NGIO switch element, with one of its ports being PCI. MT101 architecture enables system designer to build high-performance I/O system, capitalizing on advanced features of NGIO protocol (such as channel priority, reliability etc.) while using legacy I/O devices with PCI interface. The high-level system block diagram is shown on Figure 1.

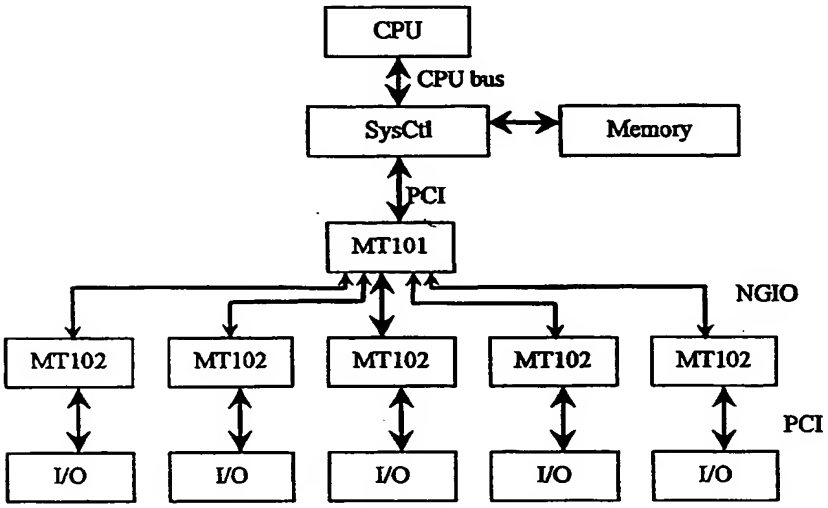


Figure 1 - MT101 system

PCI port supports 32 and 64-bit PCI bus, 33Mhz and 66Mhz. PCI-X bus support is being considered, but at this point it is out of scope of this document.

MT101 can be viewed from system PCI bus as a single p2p bridge or as multiple p2p bridges, depending how many IDSELs are connected to the PCI address lines. Maximum number of P2P bridges MT101 can be viewed as is 8. MT101/102 are not transparent to configuration SW when they implement p2p. In particular, configuration SW must explicitly set mirror MT101 configuration registers in each MT102.

Need to add high-level description of target system, whether we implement p2p bridge, whether we do it transparently to SW etc.

SW/HW architecture

The MT101/102 system provides a way to extend PCI-based system and utilize higher bandwidth by de-coupling different I/O devices, providing concurrent data transfer channels with higher bandwidth and priority-based queuing. The system contains of end-point agents (PCI unit, 8-bit CPU unit) and NGIO fabric. NGIO fabric works according to switch rules. Cell/packet sent to NGIO fabric can be squashed inside the fabric without notification. End-points (e.g. PCI unit) must assure that data transfers are completed, and issue error (interrupt) message to SW in case cell got lost in the fabric or other error occurred.

Fabric management packets can be lost in NGIO fabric, and it is solely SW responsibility to assure FMPs arrival to their destination.

The interface between NGIO world and PCI world is implemented through two basic mechanisms:

1. Explicit NGIO cell generation. MT101/102 provides 292-byte storage and control/status register that will be used by SW to construct explicitly NGIO cell and send it through the fabric. This method will mainly be used by (but not limited to) initialization SW to construct messages to be sent to the fabric.

2. Implicit translation of PCI cycles to NGIO cells. MT101 will automatically translate PCI cycles and events that need to be transferred to NGIO network, and schedule the cells/packets for transmission.

MT101 architecture also provides option to generate and forward interrupts caused by errors in NGIO fabric. Interrupts are forwarded to Master Fabric Manager through FMPs mechanism.

Interrupts' s handling

Under certain conditions MT101/102 can generate event to be delivered to SW. Examples could be link failure, interrupt or failure on secondary PCI bus etc. FMPs are used to deliver events to SW, see Events' generation and handling section for details.

Once exception occurs on the device, the respective bit in cause register is set, and if not masked – FMP is sent to Fabric Manager containing cause register in its data payload. In response to this FMP, SW will read the cause register and clear the bits. FMPSet() is used to read and clear the cause register. The FMPSet() will contain mask with bits to be cleared in cause register. Implicit FMPGet() will return the cause register after bits were cleared.

In order to assure event delivery, FMPs are used to send the message. Since FMPs can get lost in the fabric, multiple messages can be generated by both sides (HW and SW). In order to assure correct behavior regardless possible SW/HW races and possible loss of FMPs in fabric, following steps should be followed:

1. HW issues FMP send to event MAC, containing cause register as a data payload.
2. If within pre-defined period (programmable) cause register is not cleared by SW, FMP message is re-sent. If after a programmable number of re-sends no response arrived, HW will cease generating messages (severe system problem, that will be discovered and taken care of during fabric sweep).
3. Upon event delivery to SW, response FMPSet() packet should be constructed and sent to the signaling device. Cause register should be addressed and data payload should contain the value of Cause Register reported. Implied FMPGet() operation will return a masked value of Cause Register with bits sent in FMPSet() payload cleared. If returned value is not zero, it means that other interrupts arrived since original FMP generated, and SW must issue another FMPSet() until zero value is returned.
4. Only after cause register is cleared, the interrupt handling routine can start.

Data integrity

Internal data integrity

Data integrity in MT101/102 devices is assured by validating CRC in both input and output of the device. Upon receive of the cell, CRC is calculated and validated against CRC field in the cell. If mismatch encountered and end-of-cell delimiter is not EP, the *receive_error* counter of respective port is incremented. If cell transmission has not been started when error encountered, the cell will be discarded inside the MT101. If cell transmission has already been started, it will be transmitted with EP end-of-cell delimiter.

While transmitting the cell, CRC is validated again in the transmit queue. If CRC error encountered in the transmit queue and no error indication received from the receive queue, it means the cell was corrupted inside the MT101. In this case *internal_error* counter for respective transmit queue will be incremented. End-of-cell delimiter will be a 'normal' ECD.

PCI errors handling

If MT101 encounters parity error on data of the PCI cycle, it reports parity error as specified in PCI bus spec. In addition, the cell generated will be completed with EP delimiter and *PciError* counter will be incremented. If cell has not been sent to the NGIO fabric, it will be discarded. PCI target unit will not wait for acknowledge for such a cell.

PCI units (master and target) validate cells' correctness (CRC and error delimiter) before transferring cycle to the PCI bus. Corrupted cells are dropped, and error counter is incremented.

Under certain conditions PCI slave can deliver corrupted data to the PCI bus. This happens when PCI read re-try occurs right after its response arrives to the PCI slave, and data is being bypassed to the PCI bus before end of cell (delimiter, CRC etc) has arrived and cell correctness was validated (CRC, delimiter). In this case PCI unit will set a PCI data error flag in the cause register, and interrupt or SERR can be generated in response to this event.

Certain errors can be caused by erroneous configuration of the PCI port. These errors will be logged and (optionally) reported through interrupt r SERR mechanism.

NGIO errors handling

Each receive port contains a counter that counts corrupted cells arrived to the port. Cells with EP delimiter are not counted. If error encountered before cell transmit starts, it will be squashed in the MT101/102

Error reporting

All error counters of MT101 can generate interrupt to the fabric manager. If value of the counter reaches respective limit value, interrupt is generated. Setting limit to zero disables interrupts.

Minimizing errors in the network

In order to minimize flow of erroneous messages in the NGIO fabric, each receive port of MT101 should bc programmed to buffer entire cell before its forwarding to the destination queue. Note that in such a mode the latency of the communication will increase and overall bandwidth utilization will be lower. However, each receive queue will have a chance to examine cell for correctness (CRC, delimiter) before scheduling its transmission, and erroneous cells will be squashed. Although this mode is not recommended for mainstream operation, it can be handy for system debug searching for unreliable links.

Access ordering and fences

Support for ordering and fences of MT101 system is equivalent to those of NGIO. Cycles' ordering is preserved only within the same channel. Fence can be implemented on a single channel only (not on entire system). Hence, in MT101/102 system support for fence barriers originated from PCI is limited to a single channel the fence was issued to. In other words, fence will work correctly if communication path between fencing and fenced device is limited to single priority and each device has only one MAC assigned to it.

NGIO priorities

MT101/102 flexible resource management supports 4 priorities in HW. Eight NGIO priorities (zero to 7) are mapped to four HW-supported priorities as defined in NGIO spec.

LiveLock

MT101 provides capability to prevent LiveLock (when high-priority traffic blocks entirely lower-priority packets). This option is provided through LiveLock register, provided for each one of the four priority queues in each transmit queue. After queue transmitted number of cells defined in its associated LiveLock value, it 'gives up' a link for lower-priority queue for a single cell transfer. If no cells are ready for transmission in lower-priority queue, the counter will be decrement without transmission. The slot can be further 'given up' to even lower priority queue under same conditions.
Setting LiveLock value to zero disables the mechanism (e.g. cells will be transmitted in strict priority order).

Flow control

MT101 may issue flow control due to resource' overflow – either data array in the receive port is filling up or port is running out of descriptors (pointers) for arriving cells. The initial version of MT101 includes 2Kbyte data buffer and 16 pointers per each receive port.
Flow control thresholds are configurable through a pair of FC configuration registers – one for data-driven flow control and another for pointers-driven. The resource threshold left is specified per priority, and once availability of the resource exceeds the threshold, respective flow control is issued. The flow control threshold is programmed per NGIO cell priority and must be consistent with priority map of the internal priority queues as defined in NGIO configuration.
Data threshold is specified in 64-bit FCDataConfig register. Each byte defines the threshold of empty space in data array in 16-byte chunks for corresponding priority (e.g. byte0 corresponds to priority 0, byte 7 to priority 7).
Pointers' threshold is specified in 32-bit FCPointerConfig register, Each 4-bit chunk defines the threshold of empty pointers corresponding priority (e.g. chunk 0 corresponds to priority 0, chunk 7 to priority 7)
Figure 2 shows flow control configuration header.

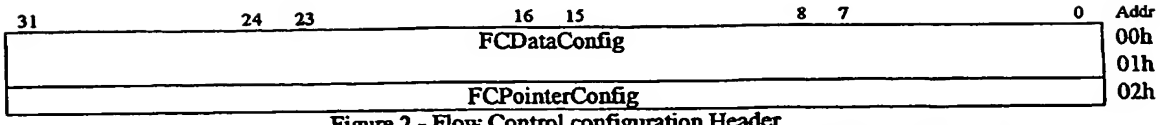


Figure 2 - Flow Control configuration Header

Data threshold is specified in 16-byte quantities of empty space in the data array. Pointers threshold specified in number of free pointers left.

PCI to NGIO interface

PCI cycles are converted to NGIO cells and sent to the fabric by PCI interface unit of MT101/102. Entire address space (memory and I/O) is divided into segments (channels), and each segment is mapped to NGIO channel (WQPs, priority, MAC etc.). PCI unit converts PCI cycles in HW to NGIO cells and sends it to the fabric. Auxiliary attributes of the channel are used to determine length and type of transfer (e.g. prefetch depth). These attributes are configurable in SW through configuration registers of MT101.

Port speed match

MT101 enables to classify ports to four different speeds. Receive/transmit arbitration logic uses this information to buffer enough cell data in the queue before start the transmission to avoid overrun on one hand and start cell transmission as soon as possible on the other hand. Table 2summarizes the rules for data buffering. Port speeds and their encoding defined in Table 1

Port Speed	Encoding
Slow	00
Medium	01
Fast	10
Very Fast	11

Table 1 - Link speed encoding

classified as Slow, Medium, Fast and Very Fast

I – immediate transmit allowed, no buffering needed

P1 – partial buffering needed, P1% of the cell should be buffered

P2 – partial buffering needed, P2% of the cell should be buffered

F – full buffering needed. Transmission cannot start till entire cell arrived to the MT101/102 device.

		Destination TxQ speed			
		Slow	Medium	Fast	VeryFast
RxQ speed	Slow	I	P1	P2	F
	Medium	I	I	P1	P2
	Fast	I	I	I	P1
	VeryFast	I	I	I	I

Table 2 - Receive/transmit port speed relation

P1 and P2 are programmable through PortSpeed register, allowed values are defined in

Value	Buffering
00	Reserved
01	¼ cell
10	½ cell
11	¾ cell

Table 3 - Port Buffering programming

Serial EPROM – initialization

MT101/102 can be initialized from microwire S-EPROM. EPROM is programmed in chunks of 3 16-bit words. The first word contains address of the control register and subsequent two words contain the data to be written to the register. On power-up MT101 sequentially reads the EPROM and loads its internal registers. The last chunk is identified by FFFFh address and its data is ignored.

MT101 enables to program S-EPROM through *ROMDATA* and *ROMSTAT* registers. These registers that can be accessed by SW from PCI, FMP or CPU interfaces.

ROMDATA register contains 16-bit address and 16-bit data to be written to the ROM. *ROMSTAT* register is used to control the S-EPROM write operation:
Bit0 – write enable. After this bit is set, Sunit writes contents of *ROMDATA*[31:16] to address specified in *ROMDATA*[15:0]. As long as this bit set, it means write has not been completed and writes to *ROMDATA* register are ignored. After write operation is completed, the bit is cleared by HW.
Bit1 – read enable. After this bit is set, Sunit reads contents (2 bytes) from the address specified in *ROMDATA*[15:0] and places it to *ROMDATA*[31:16]. This bit is cleared by HW after read has been completed. Result of reading ROMDATA register while this bit is set is undefined. Figure 3 shows template of ROMDATA and ROMSTAT registers.

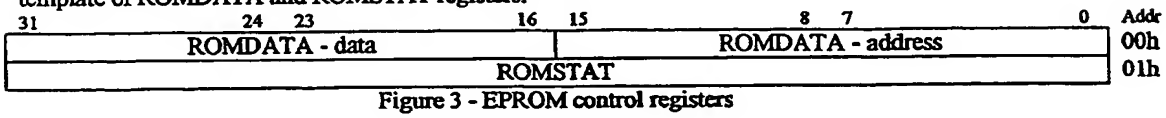


Figure 3 - EPROM control registers

PCI/NGIO system architecture

Initialization

MT101 can be initialized from HW reset, and each unit can be reset separately through *SoftReset* register. Each bit in this register asserts HW reset for different units in the device, as specified in Table 4.

Bit	Unit
0	*INIT* signal is asserted
1	PCI Target
2	PCI Master
3	NGIO Port0
4	NGIO Port1
5	NGIO Port2
6	NGIO Port3
7	NGIO Port4
8	NGIO Port5
9	NGIO Port6
10	NGIO Port7
11	FSA NGIO port
12	PCI NGIO port
13-31	reserved

Table 4 - SoftReset register

There are three phases of MT101 initialization:

Phase1 – HW reset.

After HW reset MT101 wakes up as a switch with all configuration registers loaded with their default. *INIT* signal is asserted. Phase1 is completed when HW reset is de-asserted.

Phase2 – S-EPROM sequence.

S-EPROM interface unit loads configuration registers with new values (if needed). All, some or none registers can be loaded in this phase. Upon completion of phase2 *INIT* signal will be cleared. All interfaces to the external world should be ignored until Phase2 initialization is completed. All NGIO links should be down, all PCI cycles should be delayed, all CPU cycles should be delayed. Upon completion of this phase the device is ready to operate.
The first two phases of boot will be called 'embedded configuration' in future references.

Phase3 – External initialization.

After phase2 completed, each NGIO port opens a link and PCI port starts accepting PCI cycles. External world can change configuration set in phase1 and/or phase2 using FMPs and PCI configuration cycles. During embedded configuration phases MT101 can be configured for different combinations of P2P bridges and/or multiple PCI to NGIO (P2N) bridges, as specified in Table 5. Therefore MT101 SW initialization should contain two parts, corresponding to P2P and P2N boot/configuration.

Configuration number	Number Of P2Ps	Number Of P2Ns	Total number of functions
1	0	8	8
2	1	0	1
3	1	1	2
4	1	2	3
5	1	3	4
6	1	4	5
7	1	5	6
8	1	6	7

Configuration number	Number Of P2Ps	Number Of P2Ns	Total number of functions
9	1	7	8
10	2	0	2
11	2	1	3
12	2	2	4
13	2	3	5
14	2	4	6
15	2	5	7
16	2	6	8
17	3	0	3
18	3	1	4
19	3	2	5
20	3	3	6
21	3	4	7
22	3	5	8
23	4	0	4
24	4	1	5
25	4	2	6
26	4	3	7
27	4	4	8
28	5	0	5
29	5	1	6
30	5	2	7
31	5	3	8
32	6	0	6
33	6	1	7
34	6	3	8
35	7	0	7
36	7	1	8
37	8	0	8

Table 5 - MT101 configuration combinations

In case where total number of functions is less than maximum possible (e.g. 8), it means that multiple NGIO ports are bundled to implement 'virtual' PCI bus.

Each function implements full configuration space header as defined in PCI spec. There are total eight function configuration templates in MT101, which are initialized during embedded configuration phase. P2P function will use P2P header type. The format of P2P header is defined in the P2P Bridge Architecture spec, and format of PCI device header is defined in PCI architecture spec. P2N function will use type 00h header, and its configuration fields are defined in Table 6. P2P function uses type 01h header, and its fields defined in Table 7.

Field	Value	Comment
VendorID	MLNX	Mellanox, to be defined
DeviceID	P2N	P2N, to be defined
Command	Programmed	Cannot set special cycle bit
Status	Programmed	Capabilities list bit is cleared
RevisionID	Programmed	
Class Code	P2N	Needs to be defined
Cache Line Size	Programmed	
Latency Timer	Programmed	Function per spec
Header Type	00h, 80h	Set by S-EPROM
BIST	Per spec	
BAR	Programmed	16 least-significant bits are zero

Field	Valu	Comment
Cardbus CIS Pointer	ZERO	Not implemented
Subsystem Vendor ID	Per spec	
Subsystem ID	Per spec	
Expansion ROM Base Address	Zero	Not implemented
Capabilities Pointer	Zero	Not implemented
Interrupt Line	Programmed	
Interrupt Pin	Programmed	
Min-Gnt	Programmed	
Max-Lat	Programmed	

Table 6 - Type 00h (P2N) configuration header - fields definition

Field	Value	Comment
VendorID	MLNX	Mellanox, to be defined
DeviceID	P2P	P2P, to be defined
Command	Programmed	Cannot set special cycle and VGA palette snoop bits
Status	Programmed	Capabilities list bit is cleared
Cacheline Size	Programmed	
Primary latency timer	Programmed	
Header type	01h, 81h	Set by S-EPROM
BIST	Per spec	
BAR	Programmed	16 least-significant bits are zero
Primary bus number	Programmed	
Secondary bus number	Programmed	
Subordinate bus number	Programmed	
Secondary latency timer	Programmed	Not implemented, to be programmed in MT102
I/O base	Programmed	Implemented through segments
I/O limit	Programmed	Implemented through segments
Secondary status	Programmed	Implemented through segments
Memory base	Programmed	Implemented through segments
Memory limit	Programmed	Implemented through segments
Prefetchable memory base	Programmed	Implemented through segments
Prefetchable memory limit	Programmed	Implemented through segments
Prefetchable base upper 32 bits	Programmed	Implemented through segments
I/O base upped 16 bits	Programmed	Implemented through segments
I/O limit upper 16 bits	Programmed	Implemented through segments
Capabilities pointer	Programmed	Not implemented
Expansion ROM base address	Programmed	Not implemented
Interrupt line	Programmed	
Interrupt pin	Programmed	
Bridge control	Zero	Not implemented

Table 7 - Type 01h (P2P) configuration header - fields definition

NGIO channels configuration for PCI

PCI/NGIO interface is configurable through programming Channel Headers – Target Channel Header (PCI target) and Master Channel Header (PCI master). Channel Headers are programmed as a control registers of the MT101/102. They can be programmed directly from PCI interface (on MT101) or by issuing FMPSet() operations (MT102).

Channel Headers contain all information about the NGIO channel and mapping between NGIO channel and PCI cycles' space. Each Channel Header contains address (BAR and Limit) that is mapped to this channel, type of the cycle (I/O, memory, configuration) and defines all NGIO cannel attributes (MAC, WQPN etc.) PCI cycle that was claimed by the MT101 is looked up in Channel Headers. Channel Header that covers address and type of the original PCI cycle is used to construct the NGIO packet.

NGIO channels configuration on PCI target

After PCI configuration is completed and all BARs and limits assigned, they need to be reflected to all MT devices in the system, so cycles can be routed within NGIO fabric with minimum intervention of PCI busses.
PCI target unit contains 32 PCI Target Channel Headers. Each Channel Header represents NGIO WQP. The WQP number is constructed by appending serial number (from 0 to 31) of Channel Header to upper 11 bits of *TargetWqpBase* register.
The Target Channel Header format is presented in Figure 4.

31 24	23 16	15 8	7 0	Addr
BAR				00h 01h
Limit				02h 03h
Address Map		Map Mask	*Cache line size*	Prefetch length	04h
reserved	Priority	CSN	PSN	Channel type	05h
Remote WQPN			Destination (remote) MAC		06h
RDMA-Read pending		RDMA-Read capacity	Source MAC		07h

Figure 4 – PCI Target Channel Header format

BAR, Limit fields define address space segment, as specified in PCI to PCI bridge spec.
Address Map along with Map Mask is used to re-map the most significant bits of the original address. The upper 8 bits of the new address are constructed by implementing following operation on upper 16 bits[1] of the address:
NEW_ADDRESS = (OLD_ADDRESS and MAP_MASK) or ADDRESS_MAP
Source MAC identifies MAC address of this channel.
Channel type field defines channel characteristics and is shown on Figure 5.

7 5	4 3	2 0
Reserved	Channel Type '00 – non-connected, not acknowledged '10 – connected, not acknowledged '11 – connected, acknowledged '01 – reserved	Cycle type '000 – prefetch memory '001 – non-prefetch memory '010 – I/O '011 – reserved '100 – configuration type0 '101 – configuration type1 '11x – reserved

Figure 5 - Channel type

The channel type of must be programmed to connected acknowledged channel. PCI unit does not support other channels' types.
The cycle type information is used by PCI target to construct NGIO cell, and it is used by master to issue correct command on C/BE# lines of the cycle. NGIO cells arrived to memory channel will be issued as memory read/writes. NGIO cells arrived to I/O channel will be issued as I/O reads/writes. NGIO cells arrived to configuration channel will be issued as configuration read/writes.
RDMA-read capacity field define number of oustanding read cells that can be handled by the other side of the channel.
RDMA-Read Pending field is initialized to zero, incremented each time new RDMA-Read cell cycle is sent to the channel and decrement each time RDMA-Read cell is acknowledged. If value of RDMA-Read Pending exceeds the value of RDMA-Read capacity, no new RDMA-reads are allowed to be sent to the channel till enough outstanding RDMA-Reads will be acknowledged, so number of outstanding RDMA-reads does not exceed the RDMA-Read capacity of the channel. If RDMA-read capacity field is zero, it means unlimited capacity of the far end of the channel.

[1] bits 63:56 for 64-bit address and bits 31:24 for 32-bit address

Prefetch length field is used to determine depth of prefetch (RDMA-read length) for read cycles that require more than a single PCI bus transfer (FRAME# is asserted for more than one cycle, Memory Read Multiple, Memory Read Line cycles). If this field is zero, no prefetch can be done (single-transfer cycle).

NGIO channels configuration on PCI master

PCI master unit contains 32 PCI Master Headers, each one representing NGIO WQP. The WQP number is constructed by appending serial number (from 0 to 31) of Channel Header to upper 11 bits of *MasterWqpBase* register. The Master Channel Header format is presented in Figure 6

31 24	23 16	15 8	7 0	
Priority	CSN	PSN	Channel type	00h
Remote WQPN		Remote MAC		01h

Figure 6 – PCI Master Channel Header format

P2P bridge configuration and boot

The goal of P2P algorithm is to make it as close as possible to 'native' PCI initialization, with MT101-specific code encapsulated.

In order to configure MT101 as a P2P with (optional) multiple NGIO links bundled to implement a single 'virtual' PCI bus, SW needs to implement steps summarized in Table 8. The table outlines which steps are implemented during 'native' P2P initialization and can be executed without SW modifications and which steps require MT101-specific code.

Step	Function	Comments
1	Set BAR values	**Standard SW** – sweep PCI bus, read configuration registers, set BAR value for accessing the internal registers.
2	Establish channels for MT101/102 configuration	**MT101-specific code.** Establish channels between MT101 PCI port and all MT102 PCI ports (assign MAC addresses, WQPs etc). Establishing the channels requires access to MT101 internal registers, which can be done from PCI interface for general fabric configuration. This step can be avoided by programming entire NGIO fabric from S-EPROM of MT101. Refer to SW generation of NGIO cells section.
3	Complete 'standard' system initialization	**Standard SW** – sweep entire system, assign secondary busses numbers, assign BARs to all devices in the system, assign address space mapping (base and limits) in all P2P segments.
4	Reflect configuration parameters and address mapping to all MT101/102 devices in the system	**MT101-specific code.** Establishes segment (channels) in each MT101 and MT102 by configuring channels in MT101/102 internal configuration registers.

Table 8 - P2P initialization steps

Note that – as any PCI configuration – the configuration process is recursive. If during system sweep in second step configuration SW discovers MT101 device on secondary PCI bus(es), it has to implement step1 and step 2 over again.

Secondary P2P bus can reside behind single NGIO port or can be spread between number of NGIO ports. During the third step of PCI configuration, MT101 needs to have all routing information in order to route type1 configuration cycles to the right destination. This information is provided through *PciPortConfig* registers. There are total of eight such a registers (one per each NGIO port), and their template is illustrated in Figure 7.

63 55	54 48	47 42	41 36	35 32	31 0
Secondary bus number	*Subordinate bus number*	Type1 channel #	Type0 channel #	Configuration template number	IDSEL mask

Figure 7 - *PciPortConfig* register template

MAC Address field specifies MAC address of PCI master in MT102 device.

Configuration template number defines which configuration template (one out of eight) this port belongs to.

IDSEL mask field is an OR of decoded device numbers (IDSEL) of the secondary bus devices that are mapped to this NGIO port[2].
Fields specified in italic are alias from the configuration template, defined by configuration template number field in the register[3].
When PCI unit of MT101 observes the configuration type1 cycle on PCI bus, it looks up the *PciPortConfig* registers to identify whether this cycle should be claimed by MT101. If Bus Number field of the type1 cycle belongs to the range covered by MT101 (e.g. it fails in one of the secondary bus ranges covered by MT101 functions), it claims the cycle.
If Bus Number field equals to one of the secondary bus numbers covered by MT101, the cycle is converted to type0 configuration cycle, and NGIO RDMA cell constructed. The destination port is onewhose Secondary Bus Number filed in *PciPortConfig* register matches Bus Number field of original type1 transaction, and decoded value of Device Number field in original PCI cycle is not masked (nullified) by IDSEL Mask value in *PciPortConfig* register.
If Bus Number field equals in type1 transaction belongs to the range covered by MT101, but not equal to any of its secondary bus numbers, MT101 generates NGIO RDMA cell with type1 configuration. Destination is determined from *PciPortConfig* registers using Secondary Bus Number and Subordinate Bus Number fields.
The resulting cell is sent to the channel whose number is specified in *Type0 Channel #* field of the *PciPortConfig* register for type0 configuration cell, and to Type1 Channel # field for type1 configuration cells. The segment (channel) registers on both sides of the channel should be programmed appropriately to assure correct operation.

31 — 24	23 — 16	15 — 8	7 — 0	
Port0 PciPortConfig				00h
				01h
Port1 PciPortConfig				02h
				03h
Port2 PciPortConfig				04h
				05h
Port3 PciPortConfig				06h
				07h
Port4 PciPortConfig				08h
				09h
Port5 PciPortConfig				0Ah
				0Bh
Port6 PciPortConfig				0Ch
				0Dh
Port7 PciPortConfig				0Eh
				0Fh

Figure 8 - P2P configuration registers summary (*P2PConfig*)

[2] If secondary PCI bus is mapped to a single NGIO port this register corresponds to, all bits should be set in IDSEL field
[3] Note that IDSEL mask and Configuration Template Number fields in *PciPortConfig* register are filled in during embedded configuration. The second step in PCI configuration (Table 8) is necessary in order to assign MAC addresses for PCI masters in MT102s and fill in MAC Address field in *PciPortConfig* registers. If MAC addresses can be assigned during embedded configuration phase, the second step can be skipped and PCI initialization SW can run without interception.

PCI/NGIO interface

PCI cycles conversion to NGIO cells

After all channels are configured as specified in previous sections, MT101/102 can automatically convert PCI cycles to NGIO cells and send them to the NGIO fabric. In addition, NGIO cells that arrive to PCI destination channels will be automatically converted to PCI cycles.

Once PCI slave decoded PCI cycle that maps to its address space, it converts it to the NGIO cell obeying NGIO rules. Table 9 summarizes PCI CMD to NGIO translation. PCI unit issues only RDMA-read and RDMA-write cells to the NGIO fabric as valid NGIO cells. For PCI destinations, the command in PCI cycle will be determined by the PCI master based on channels attributes cell arrived to.

CMD	Command	NGIO cell
0000	INTA	None
0001	Special cycle	None
0010	I/O Read	RDMA-read, length as specified in original cycle
0011	I/O Write	RDMA-write, length as specified in original cycle
0100	Reserved	None
0101	Reserved	None
0110	Memory Read	RDMA-read, length according to Prefetch Length field in Target Channel Header format
0111	Memory Write	RDMA-write, length as specified in original cycle
1000	Reserved	None
1001	Reserved	None
1010	Configuration Read	RDMA-read, length – 4 bytes
1011	Configuration Write	RDMA-write, length – 4 bytes
1100	Memory Read Multiple	RDMA-read, length according to Prefetch Length field in the Target Channel Header
1101	Dual Address Cycle	None
1110	Memory Read Line	RDMA-read, length according to Prefetch Length field in the Target Channel Header
1111	Memory Write and Invalidate	RDMA-write

Table 9 - PCI cycle CMD to NGIO cell translation

Cell will be legal NGIO cell:

1. MAC, port number, priority, PSN, MTH and WQ pair (source and destination)are taken from respective channel (BAR segment).
2. The NGIO data access must be consecutive string of bytes. If not all byte enables were active in the PCI cycle, slave must split this cycle to multiple NGIO cells and assure that each one contains consecutive byte string (read or writes)

Following are the rules for RDMA-read cells generation:

1. For read-multiple PCI cycles slave generates single cell.
2. For read-multiple PCI cycles length of RDMA-read is taken from configuration register associated with the channel the read is targeted to.
3. Length of the read must obey PCI rules for data prefetch.

If RDMA-read response does not arrive within time period set in MemLifeTime register, interrupt is generated according to the INT/SERR mask register configuration or target-abort response generated to the next re-try of original read. The transaction is removed from the pending transactions queue.

RDMA-write generation rules:

1. Length or multiple writes is limited to 128 bytes (? Maybe just unlimited? Just based on the buffer availability? What about alignment?)
2. For posted writes TRDY# is returned immediately and RDMA-write is sent to the target. If not acknowledged within time specified in MemLifeTime register, interrupt is generated.
3. For non-posted writes cycle is stopped (re-try), original data is kept and RDMA-write cell sent to the NGIO network. When write originator re-issues the cycle after this cell was acknowledged, PCI slave compares all cycle attributes (address, data, byte enables etc.) with original cycle, and if match occurred – TRDY# is returned to the originator. If after RDMA-write acknowledge original cycle was

not re-issued within pre-defined period, PCI slave either squashes original request or generates interrupt on the PCI bus.

NGIO cells conversion to PCI cycles

NGIO cells that arrive to the PCI unit will be converted to the PCI cycles. The command driven on C/BE# lines of the PCI bus will be in accordance to NGIO cell type and channels attributes specified in Channel Type field of the Master Channel Header.

Cycle Type	RDMA-read	RDMA-write
'00, (prefetch memory)	'0110 or '1100 (depending on length)	'0111
'01 (non-prefetch memory)	'0110	'0111
'10 (I/O)	'0010	'0011
'11 (configuration)	'1010	'1011

Table 10 - NGIO cells to PCI cycles translation

PCI cycles generation from NGIO interface

MT101 architecture provides a way to generate cycles on PCI bus through programming *PciSpecialCycles* registers. These registers are accessible through FMPSet() operation, hereby enabling generate PCI special cycles from NGIO interface. Every PCI cycle can be generated through this mechanism. Data transfer length is limited to 8 bytes.

The mechanism is provided through *PciCycle* control register. Figure 9 illustrates its format and fields.

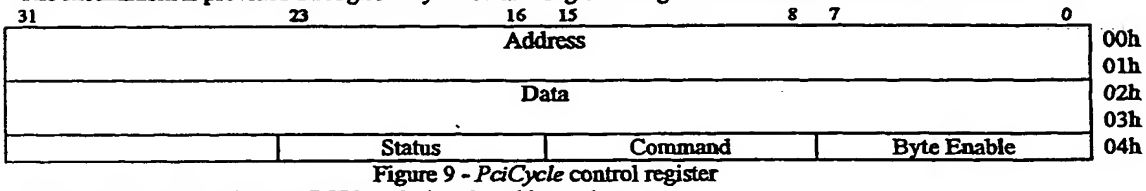


Figure 9 - *PciCycle* control register

Address field to be driven on PCI bus during the address phase.

Data field contains data to be driven to PCI bus during data phase of the write cycles or is a target for a data read in read cycles.

Byte Enable filed contains value to be driven on BE# lines during Byte Enable phase

Command field contains command to be driven on PCI bus on C/BE# field, byte enables and control bits and handshake status/control bits. The lower-order bits of Command field represent value to be driven on C/BE# lines. Bit8 of Command field is set by SW, indicating that Address, Data, Byte Enables and CMD values are written and cycle should be driven to PCI bus. After PCI transaction has been completed, HW clears bit4, and Status field contains information about cycles' completion status.

Encode	Status
'1xx	Cycle in progress
'000	Normal completion of the cycle
'001	Re-try
'010	Master-abort
'011	Target-abort

Table 11 - PCI cycles completion status

SW generation of NGIO cells

MT101 provides a capability to generate and accept NGIO cell from NGIO fabric. This is capability is provided through System NGIO Port, which contains two 292-byte data structures accessible through as MT101 internal registers and control/status register. The first structure – *OutBoundCell* – is written by SW through internal registers' access mechanism. The data written should be a valid format of NGIO cell. After data is written to *OutBoundCell*, a *outbound_full* bit set in *SystemPortDoorbell* register, which initiates a send process. After cell has been sent to the NGIO fabric, HW clears *outbound_full* bit in *SystemPortDoorbell* register, signaling that *OutBoundCell* is empty, and new cell can be filled in. The second 292-bit data structure – *InBoundCell* – is used to accept new cells from the fabric. Cells targeted to System Port are stored in *InBoundCell* and *inbound_full* bit is set in *SystemPortDoorbell* register, indicating that new cell arrived. SW reads the data structure and clears the *inbound_full* bit, enabling new

cell to arrive. As long as *inbound_full* bit is set, new cell that arrives to System Port will be dropped, and *dropped_cell* counter in *SystemPortDoorbell* register will be incremented, so SW can have a notice of dropped cells. Figure 10 illustrates System Port configuration registers.

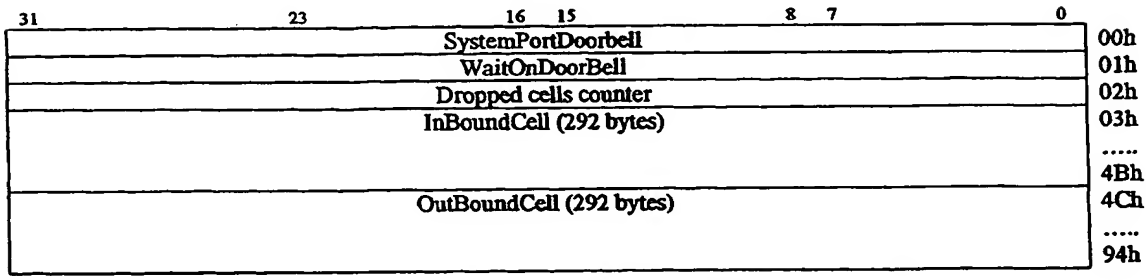


Figure 10 - System Port configuration registers

SystemPortDoorbell register is shown on Figure 11.

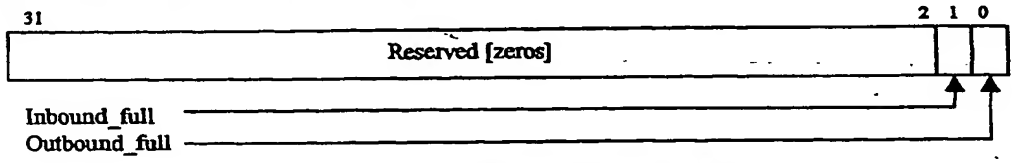


Figure 11 - SystemPortDoorbell register

WaitOnDoorBell register is used to wait till SystemPortDoorbell register is assigned in HW value that is written to WaitOnDoorBell register. On write to WaitOnDoorBell register, HW does not return acknowledge until value of SystemPortDoorbell register does not match value written to the WaitOnDoorBell, and no more configuration register access can start. This mechanism will enable to initiate entire NGIO fabric from S-EPROM. Note that careless use of this mechanism can hang the system, as access to all control registers will be blocked forever. This functionality is enabled only for accesses originated by S-EPROM.

NGIO boot

MT101/102 can be booted through NGIO boot mechanism, as specified in NGIO spec. Although MT101/102 does not implement HCA function to full extend, its architecture provides the capability to boot the entire system through NGIO interface. This can be done by generating NGIO cells explicitly (refer to SW generation of NGIO cells section).

NGIO cells arriving to FSA can alter control registers of the MT101/MT102. It will usually be Priority15 messages, although messages with other priorities arriving to the FSA will be treated as configuration messages as well.

Configuration messages are messages whose destination MAC matches FSA MAC address, CMD field is FMPGet() or FMPSet(). Configuration channels are non-connected, no acknowledge will be sent back to the request queue except implicit FMPGet() in response to FMPSet() message.

The data payload of the cell contains address of internal register to be accessed, command (read, write) and number of registers to be read. Configuration message can be either direct route or MAC-addressed. Data Payload format of the direct-routed NGIO configuration message is presented in Figure 12 and Data Payload format of the MAC-routed configuration message is presented in Figure 13.

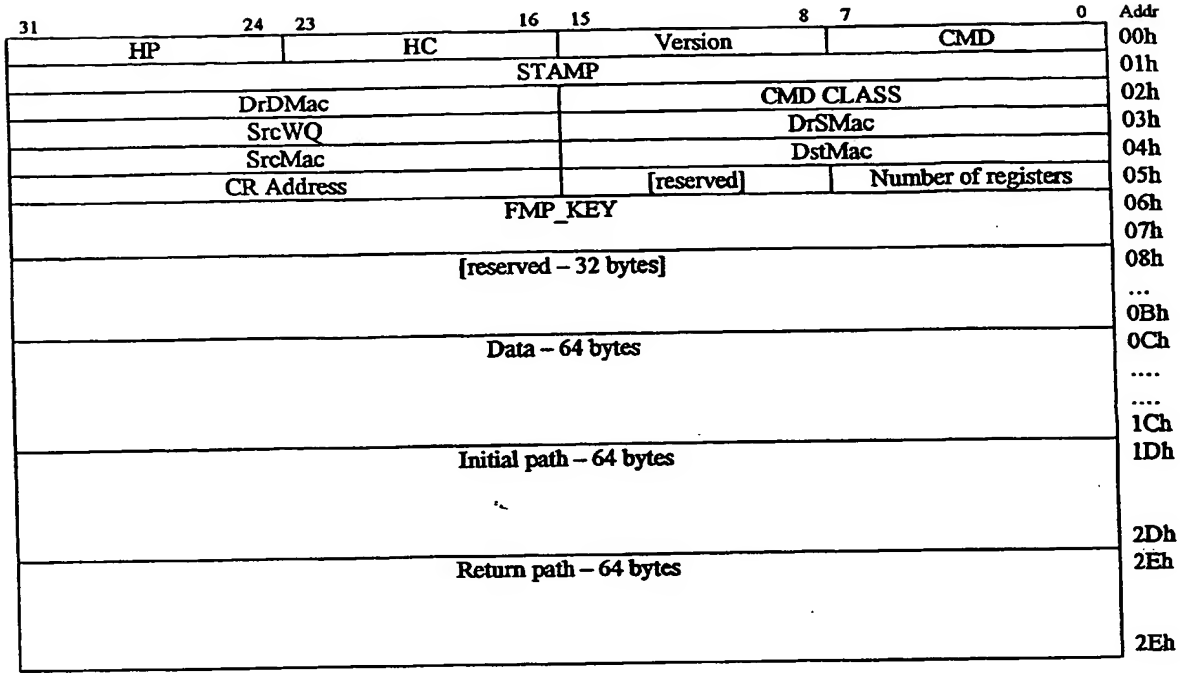


Figure 12 – Direct-route Configuration message Data Payload format

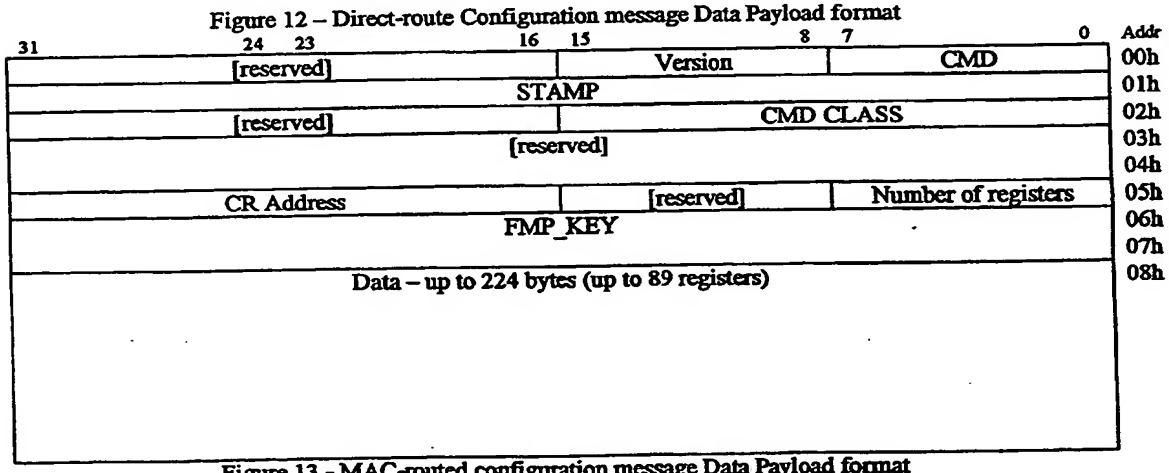


Figure 13 - MAC-routed configuration message Data Payload format

CMD field specifies whether it is CR read (FMPGet()) or CR Write (FMPSet()) operation.
Number of registers field specifies number of registers to be accessed. Note that with direct-routed message only 16 registers can be accessed by a single message.
MAC-routed FMPGet() message does not need to contain 224 bytes of data; the response message should append data as required by the number of registers accessed.

MT101 Configuration Summary

This section summarizes configuration registers of MT101. Some registers are specified as alias to other registers. This means that writing to the register these registers are alias to will update the register. However, this value can be overridden by writing to explicit address of the destination register. This

enables an easy way for debug and makes life easier for spin-off products (modularity – in both HW and SW).

Global MT101 configuration (FSA)

Global MT101 configuration information which defines operation of entire MT101 device. Figure 14 defines Global MT101 configuration header. This header resides in FSA and managed by FSA. Different fields of the register can be altered (or not altered) by FMPs, refer to NGIO spec – FSI section.

31 – 24	23 – 16	15 – 8	7 – 0	Addr	
HostGUID				00h 01h 02h 03h	DevInfo COD, index0
PmState	FmpVersion	NumPort	DevType	04h	
CapabilityMask				05h	
MembershipId		DiagCode		06h	
SNMP_WQ		SNMP_MAC		07h	
DeviceID		VendorID		08h	
Revision				09h	
MlxLevel	BootPort	BootMac		0Ah	
DeviceString [16 registers]				0Bh 1Ah	DevInfo COD, index1
DiagData		NextIndex		1Bh	DevInfo COD, index3
DiagData [15 registers]				... ... 2Ah	
FmKey				2Bh 2Ch 2Dh	PortInfo COD
TimeOut		ActiveFm			
Reserved			Protbit	2Eh	
SwitchCellLife		FDBCap		2Fh	SwitchInfo COD
PerfSigWQ		LifeTimeValue		30h	
reserved	NumQs	PriMap	MgtPort	31h	
FDB [64 registers, 256 FDB entries]				32h ... 71h	FDB
Config. (Table 5)	Default Port #	PortSpeed (2 bits per port, Table 1)		72h	
Flow Control Configuration (Figure 2) [3 registers]				73h 74h 75h	

Figure 14 - Global MT101 configuration Header

NGIO port configuration

Figure 15 illustrates NGIO port configuration Header. These registers exist in every 'native' NGIO port (e.g. excluding ports that serve FSA, PCI etc.). Fields specified in italic are alias to respective filed in the Global Configuration Header (Figure 14)

31 – 24	23 – 16	15 – 8	7 – 0	Addr
Port performance management header (Figure 22) [28 registers]				00h 1Bh

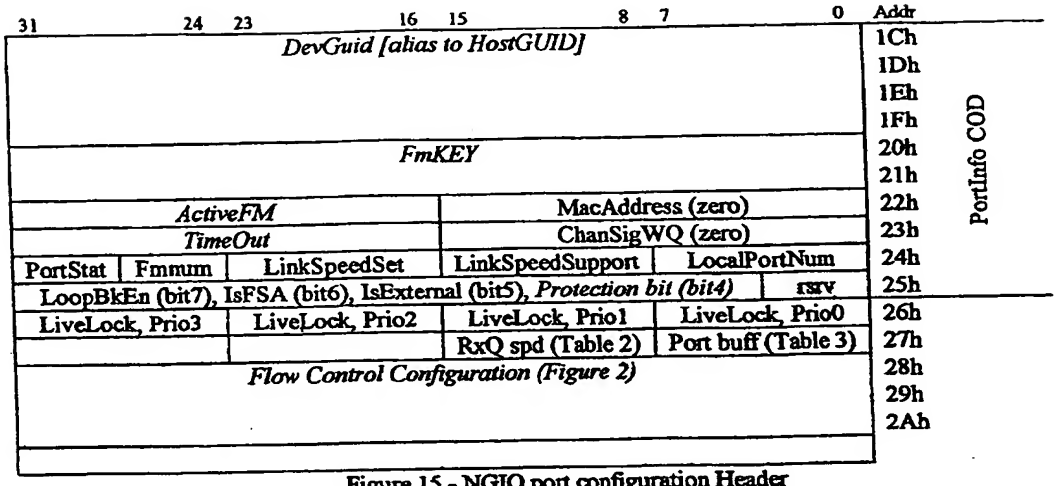


Figure 15 - NGIO port configuration Header

PCI configuration

Figure 16 presents summary of PCI Configuration Header

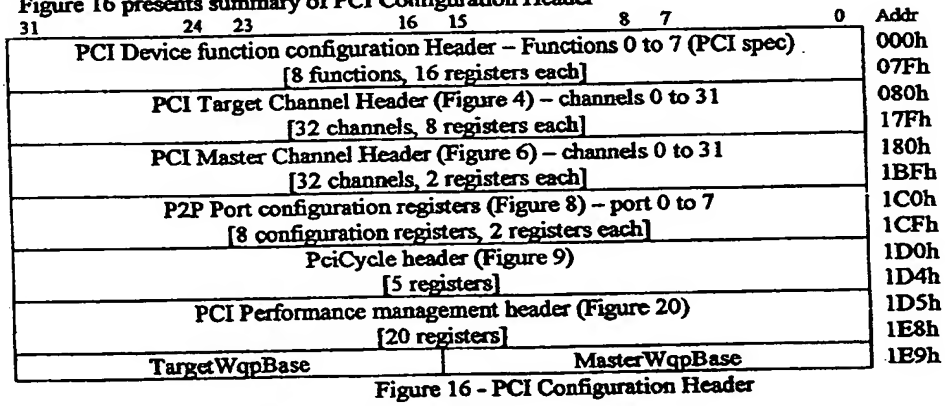


Figure 16 - PCI Configuration Header

Miscellaneous configuration registers

Figure 17 shows miscellaneous configuration registers of MT101

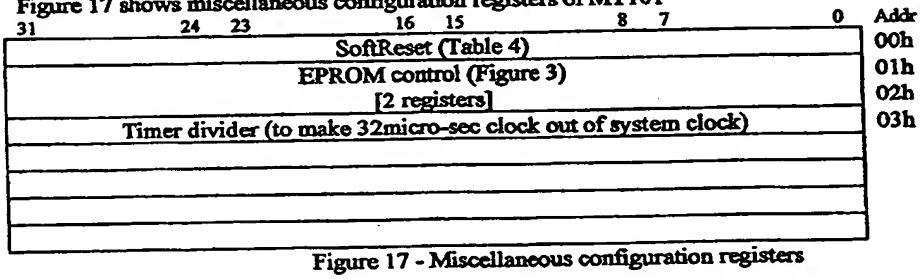
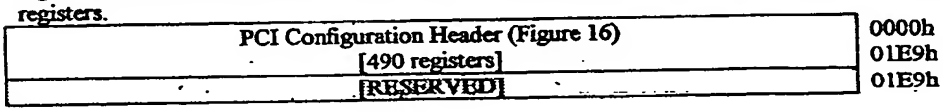


Figure 17 - Miscellaneous configuration registers

Configuration space summary

Figure 18 shows configuration space of MT101, including address assignments of the configuration registers.



for future PCI expansion ☺	03FFh
NGIO port configuration – port 0 to 7 (Figure 15) [8 ports, 64 possible registers each]	0400h 05FFh
RESERVED For future NGIO ports expansion ☺	0600h 07FFh
System Port configuration (Figure 10) [149 registers]	0800h 0894h
FSA Performance Management Header, event generation part (Figure 24) [13 registers]	0895h 08A1h
FSA Performance Management Header – recipient side (Figure 26) [24 registers]	08A2h 08B9h
MT101 Global Configuration registers (Figure 14) [115? Registers]	08BAh 092Ch
[RESERVED] For future global expansions ☺	092Dh 09FFh
Miscellaneous configuration registers (Figure 17) [??? Registers]	0A00h 0AFFh

Figure 18 - MT101 configuration space summary

Events' generation and handling

MT101 and MT102 can generate and/or forward events to be delivered to Fabric Manager. This section specifies in details the mechanism of event's generation and delivery.

Event's generation

Once event is generated, it is delivered to fabric manager by FMP_TRP_REQ_MSG. The message is constructed and sent by FSA unit of the MT101. If fabric manager interfaces with MT101/102 network through PCI or CPU endpoints, there are two options provided for message delivery:

1. MT101/102 end-point will write the incoming FMP cell to memory and ring the doorbell.
2. MT101/102 end-point will keep the arrived FMP cell in internal register and ring the doorbell

In first case multiple FMP trap messages can arrive before the first one is handled by SW. It is SW responsibility to avoid FMP trap stack overflow (e.g. SW should poll the FMP traps stack). In second case only one (first) trap message will be kept until read by SW. In both cases HW interrupt can optionally be asserted upon new trap arrival (doorbell)

HW interface for event delivery

All events are delivered to SW through FMPs. FSA is exclusively responsible to deliver event to SW, implementing following steps:

1. Set appropriate bits in Cause Register
2. Construct FMP using EventFMPTemplate upon event request generated by HW.
3. Send this FMP to Fabric Manager
4. Wait for FMP acknowledging the event (clearing bits in Cause Register)
5. Re-send event FMP in case interrupt acknowledge FMP did not arrive within pre-defined time
6. Cease re-sends after ResendCount exceeded.

Figure 19 defines the Event FMP format. Shaded fields are taken from EventFMPTemplate register. Reserved fields are filled with '0

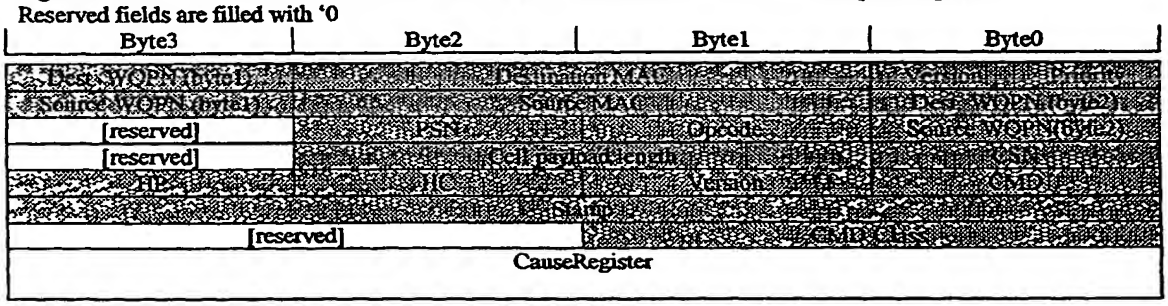


Figure 19 - event FMP format

Control and status summary – errors and performance monitoring

Link errors are handled through Error Headers depicted in figures below. The definition is derived from NGIO Performance Management concept. Refer to Switch Spec, chapter 6 for more details and explanations,

PCI Performance Management Header

PCI port can encounter additional errors due to following reasons:

1. BAR/Limit range mismatch – part of address space defined in function template is not covered by channels
2. BAR/Limit range mismatch – overlap in channels' address space
3. IDSEL/secondary bus mismatch – device number is not covered by IDSEL mask
4. IDSEL/secondary bus mismatch – device number is covered by more than one channel

Bugs in configuration SW will cause these errors. If such an error encountered, the respective cell is dropped and error occurrence is logged in PCI Error Counter.

31 24	23 16	15 8	7 0	Addr
NGIO Inbound Data Error counter				00h
NGIO Inbound Data Error Limit				01h
PCI Error counter				02h
PCI Error Limit				03h
PCI Error address				04h
				05h
NACK posted write counter				06h
Posted writes NACK Limit				07h
Sequence Error counter				08h
Sequence Error limit				09h
NGIO response timeout				0Ah
PCI re-try timeout				10h
Reserved		Channel Header		11h
PCI Error Cause				12h
PCI Error Mask				13h

Figure 20 - PCI Performance Management Header

Inbound counters count number of erroneous cells (EP delimiter or wrong CRC) received from NGIO fabric.

PCI error counters count number of PCI errors reported (e.g. parity error, configuration error etc.). The address of the cycle that resulted in error is stored in PCI Bogus address field.

Channel Header field contains the number of channel header that last encountered an error (data error, sequence error, NACK, timeout on reads etc.)

Error counters are cleared at reset. Each time error occurs, respective counter is incremented. If counter reaches respective limit value, the respective bit is set in PCI Error Cause register and event is generated if is enabled by the PCI Error Mask (respective bit is not cleared in PCI Error Mask register).

PCI Error Cause Register is defined in Figure 21

Bit	Cause
0	NGIO Inbound Error
1	PCI Error
2	NACK posted write
3	Sequence error
4	NGIO response timeout
5	PCI re-try timeout
6-31	Reserved

Figure 21 - PCI Error Cause register

LstChanErrWQ and LstSeqErrWQ are alias to respective values in TcaErrorInfo COD, Index0 and ChanSeqErr is alias to TcaErrorInfo COD, Index1. Other parameters of TcaErrorInfo COD can be computed off the values in PCI Error Header.

NGIO port performance management header

The NGIO port performance management and error reporting is defined in compliance to Performance Monitoring definition (Switch, section 6). Figure 22 defines the NGIO Performance Management Header

31 24	23 16	15 8	7 0	
PortTxOctets				00h
PortRxOctets				01h
PortTxCells				02h
PortRxCells				03h
PortRxErrors				04h
PortRxCellDiscards				05h
PortTxCellDiscards				06h
PortRxCellsTooShortErr				07h
PortRxCellsTooLongErr				08h
PortRxCellsCRCErr				09h
PortRxCellsDisparityErr				0Ah
PortRxCellsEncodeErr				0Bh
PortRxPriError				0Ch
PortRxDestRxPort				0Dh
PortTxLifetimeErr				0Eh
PortTxExcessFCErr				0Fh
PortTxActiveErr				10h
PortTxOctets Limit				11h
PortRxOctets Limit				12h
PortTxCells Limit				13h
PortRxCells Limit				14h
PortRxErrors Limit				15h
PortRxCellDiscards Limit				16h
PortTxCellDiscards Limit				17h
Internal Error counter				18h
Internal Error Limit				19h
NGIO Port Error Cause				1Ah
NGIO Port Error Mask				1Bh

Figure 22 - NGIO Port Performance Management Header

Italics fields defined in Switch Spec, section 6.

Internal Error counter counts error generated inside the MT101, as described in Internal data integrity section.

Port Error Cause register logs events by setting appropriate bit and event is generated if not masked by respective bit in the Event Mask register

Bit	Cause
0	PortRxOctets
1	PortTxCells
2	PortRxCells
3	PortRxErrors
4	PortRxErrors
5	PortRxCellDiscards
6	PortTxCellDiscards
7	PortTxLifetimeErr
8	PortTxExcessFCErr
9	PortTxActiveErr
10	Internal Error
7-31	reserved

Figure 23 - NGIO port error cause register

FSA performance management header – event generation side

FSA consolidates all events generated in the MT101, constructs a combined Cause Register, constructs FMP event message and sends it to the Active Fabric Manager MAC.

Figure 24 defines the FSA Performance Header – the event generation part

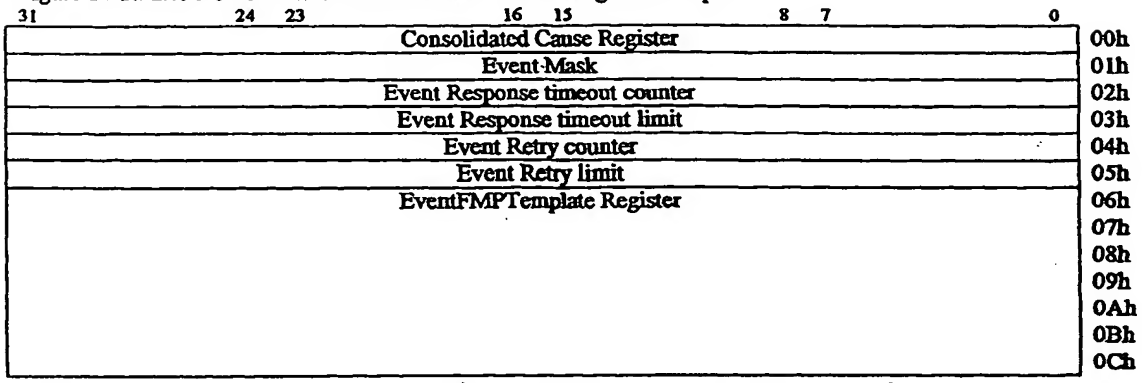


Figure 24 – FSA Performance Management Header, event generation part

Consolidated Cause Register includes information about all events that happened in this device. Event's generation (FMP message) can be masked by programming Event Mask register – event is generated only if corresponding bit in Event Mask register is set.

Bit	Cause	Bit	Cause
0	NGIO link down	16	Trap RDMA-Write timeout/NACK
1	NGIO link up	17	
2	NGIO RxQ err limit exceeded	18	
3	NGIO TxQ err limit exceeded	19	
4		20	
5		21	
6		22	
7		23	
8	PCI sequence error	24	NGIO Octets/Cells limit (either)
9	PCI RD/non-post WR bad response	25	
10	PCI posted WR bad response	26	
11	PCI interrupt INT	27	
12	PCI bus error	28	
13		29	
14		30	
15		31	

Table 12 - Consolidated Cause Register

EventFMPTemplate register is defined in Figure 25.

31 24	23 16	15 8	7 0	
Dest. WQPN (byte1)	*Destination MAC*		*Version* / *Priority*	00h
Source WQPN (byte1)	*Source MAC*		*Dest. WQPN (byte2)*	01h
[reserved]	PSN	Opcode [04h]	Source WQPN(byte2)	02h
[reserved]	Cell payload length		CSN	03h
HP [0]	HC [0]	Version [0]	CMD [04h]	04h
Stamp [0]				05h
DrDMAC [0]		CMD Class [0]		06h
SrCWQ [0]		DrSMac [0]		07h
SrcMac		*DstMac*	COD	08h
COD_IDX		[reserved]		09h
FMP_KEY				0Ah
				0Bh

Figure 25 - EventFMPTemplate register

FSA performance management header – event recipient side

Once FMP is generated, it is forwarded to Active FM MAC address. In MT101 systems FSA of the MT101 can serve as a destination of the Event Message. It provides basic HW hooks for SW interface – stores recipient message in internal register, can optionally translate it to RDMA-write packet and forward it to port with memory (e.g. PCI or 8-bit CPU). It also can optionally assert INT or SERR output of the MT101.

31 24	23 16	15 8	7 0	
Received Cause Register				00h
INT Mask				01h
SERR Mask				02h
RDMA-Write Mask				03h
Memory Stack stride				04h
Reserved			RDMA-Write priority	05h
RDMA-Write destination WQPN		RDMA-Write Destination MAC		06h
RDMA-Write source WQPN		*RDMA-write source MAC*		07h
Reserved		CSN	PSN	08h
RDMA-Write VA/MH				09h
				0Ah
RDMA-Write MH				0Bh
RDMA-Write Response timeout counter				0Ch
RDMA-Write Response timeout limit				0Dh
RDMA-Write Retry counter				0Eh
RDMA-Write Retry limit				0Fh
FMPTrap Door Bell register				10h
FMPTrapCell register [292 bytes? Seems too much]				11h
				....
				....
				....
				....
				....
				????

Figure 26 – FSA Performance Management Header – recipient side

Upon FMPTrap() message arrival, FSA checks whether it is a destination for this message by examining destination MAC address. If destination MAC address matches its own address, FSA extracts Cause Register from the FMPTrap() message and stores it in the FSA Performance Management Header. The entire message is stored in the FMPTrapCell register, sets bit0 in FMPTrap DoorBell register. If interrupt or SERR is enabled by the respective mask in the FSA Performance Header, FSA asserts INT or SERR pins.

If Memory Event Stack is enabled by respective bit in the RDMA-Write Mask, FSA generates RDMA-Write message with MAC header fields specified in the FSA Performance Management Header (09h-0Bh).
The data payload of RDMA-Write should contain MAC header of the original FMPTrap() message and cause register. The address pointer (RDMA-Write VA register) should be incremented by the value stored in Memory Stack Stride (sign-extended), so it will be ready for the next message.
If RDMA-Write was not acknowledged within RDMA-Write timeout limit after RDMA-Write retries limit or it was NACK'ed, FSA sets Trap-RDMA-Write timeout/NACK bit in the Consolidated Cause Register of the FSA Performance Management Header (event generation part). This may result is sending the FMPTrap() message to the destination as specified in the EventFMPTemplate of this device.
In order to avoid endless loops, RDMA-Write should be masked for Trap-RDMA-Write timeout event if destination of the FMPTrap() is the same FSA that issued the RDMA-Write.
Normally, Trap RDMA-Write messages should be sent with priority15 (FMPs) to non-connected destination, and acknowledge would be clearing Cause Register by SW. However, it is possible to send this message to connected/acknowledged channel (that should be configured ahead on the destination side).

MT101 and MT102 overview

MT101 architecture is a baseline architecture that is implemented in multiple products, first being MT101 and MT102. MT101 block diagram is shown on Figure 27, and MT102 block diagram is shown on Figure 28. As could be noticed from these figures, MT102 is a subset of MT101 component. MT101 internal architecture design is targeted to simplify MT102 design. The inter-unit protocols have no notion about number of units and their nature. Global chip resources are not limited to any fixed number of NGIO or PCI ports. This document will describe the MT101 architecture.

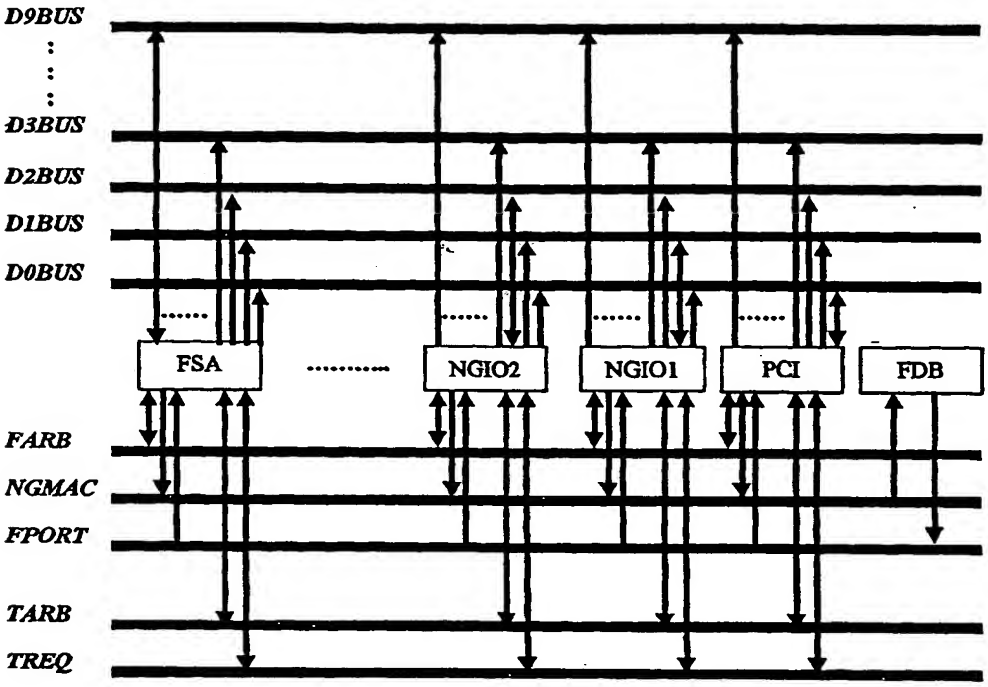


Figure 27 - MT101 block diagram

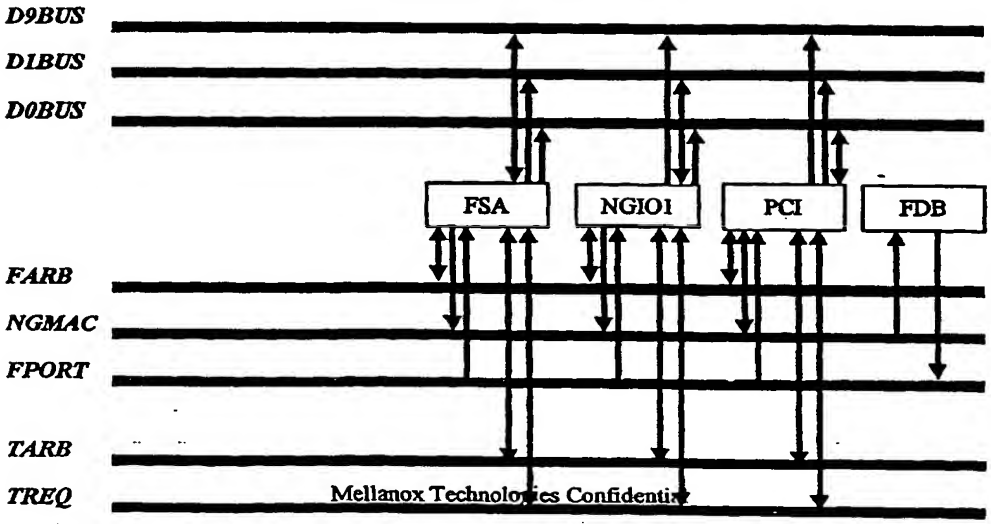


Figure 28 - MT101 block diagram

There are three groups of the busses:
1. *FBUS* group (including *FARB*, *FPORT* and *NGMAC*). This group implements phase1 of the transaction protocol.
2. *TRQ* group (including *TARB* and *TREQ*). This group implements phase2 of the transaction protocol
3. *DBUS* group, which includes data busses. This group implements phase3 of the transaction protocol.

Block description

NGIO unit is NGIO port, implementing NGIO receive and transmit queues. Detailed description of NGIO unit is available in NGIO port external definition and requirements.
FSA unit is a Fabric Service Agent unit, responsible for perform all fabric management functions of MT101 device. Detailed description of FSA unit is available in FSA unit external definition chapter.
PCI unit is PCI port, responsible to accept PCI cycles and route them to NGIO network. It is also responsible to transfer NGIO network requests for PCI resources to PCI bus. Detailed description of PCI unit is available in PCI port external definition and requirements.
Each unit is has a data bus associated with it, which is used to send data to the unit for transmission.
Multiple ports can be mapped to same unit.

Data transactions

Arbitration protocol overview

This section describes arbitration protocol used by all busses. Distributed arbitration will be deployed in on all MT101 busses, which avoids long round-trip paths and enables easy scalability of the architecture.
Arbiter reference design is available in Appendix B – reference designs (common blocks). Same arbiter design should be used (instantiated) in each unit that arbitrates for a particular bus.
General connection for a bus to be arbitrated is shown at the Figure 29 below. The scheme assumes N devices that connected to the same bus ***BUS*** and they use ***ARB[n-1:0]*** signals for arbitration.

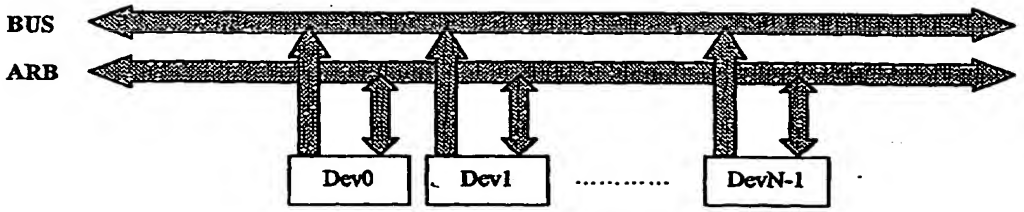


Figure 29 - bus arbitration - general case

Each device *m* has an ***ARBm*** line associated with it. This is the line it drives to active state if it wants to acquire ownership on BUS. The position of *m* bit in ***ARB[n-1:0]*** signal identifies priority of this device in arbitration. The higher *m* is, the higher is priority of the device. Protocol assures that if two devices – *m* and *n* (*m*>*n*) requested ownership of the bus at the same clock, device *m* will acquire ownership, and device *n* will give up, as priority *n* is lower than *m*.
At the beginning of arbitration cycle, every device puts its request for ***BUS*** on respective ***ARBi*** line (where *i* is priority of the device at that point). At the end of this cycle each device observes entire ***ARB*** signals. If no higher priority request was placed on ***ARB*** signals, it means the arbitrating device granted ownership of ***BUS*** and can drive it next cycle. If devices notices request of higher priority than is own placed on ARB signals, it means arbitration of is failed and ***BUS*** ownership was not granted. The device can arbitrate again in the next cycle. Arbitration protocol is fully pipelined. Figure 5 illustrates arbitration between 3 devices.

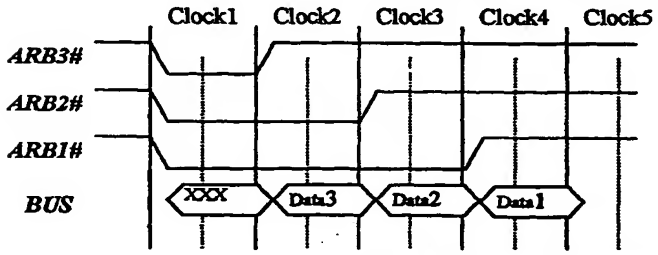


Figure 30 - arbitration example

In clock1 all 3 devices (priority 1,2 and 3) place their request for ***BUS*** arbitration. At the end of clock1 by observing ***ARB*** lines, device1 and device2 noticed that higher priority device (device3) requested bus in the same clock (***ARB3#*** asserted), and therefore their arbitration failed. Device3 assumes ownership of the ***BUS*** for the next cycle.
In clock2 device3 drives his data on ***BUS*** lines, and devices 1 and 2 arbitrate again on the ***BUS***. Device1 notices that higher priority device (device2) requests the bus and gives up. Device2 does not see any higher priority arbitration requests, and therefore it is granted BUS ownership for the next cycle.
In clock3 device2 drives his data on ***BUS*** lines, and device1 arbitrates again. This time no higher priority requests posted, therefore device1 is granted the bus and drives its data on clock4

Static priority

Using arbitration protocol described above, static priority between devices can be achieved by static assignment of ***ARB*** lines for each device. Thus, if device *m* always drive ***ARBm#*** line in arbitration cycle, and device *n* drives ***ARBn#*** (*m>n*), this means device m will always have higher arbitration priority than device *n*.

Cyclic priority

Static priority has a drawback that low-priority device can be starved. In order to prevent starving, cyclic priority assignment will be used when appropriate.
Unlike static assignment of ARB line per device, in cyclic priority ARB line assignment is changed every clock in cyclic manner. Thus, at any given time the arbitration priority of all devices is random, which effectively provides equal priority for each device. In order to avoid collisions, each arbiter is initialized with distinct priority and all arbiters are fully synchronized.

Data transfer phases

Upon receiving of NGIO cell (or upon translating PCI cycle to NGIO cell), the data transfer starts. Each data transfer goes through following steps (phases):

Phase1 – MAC translation

Receive queue extracts MAC address from accepted cell and translates it to port address using FDB table. It is possible to have a local copy (or cache) of FDB table or inquire FDB unit for translation. ***FBUS*** implements this phase.

Phase2 – post transmit request

Receive queue arbitrates request bus (***TREQ***) and posts transmit request to the transmit queue.

Phase3 – transfer data to transmit queue

Transmit queue requests data transfer from receive queue, and data is transferred on data part of ***DiBUS***. ***TRQ*** and ***DiBUS*** busses implement this phase.
Upon completion of one phase, the transaction may wait in the queue for unlimited time till its next phase is scheduled.
In heavy load environment each transaction is expected to go through all three distinct phases. In light load performance optimizations are made to boost transfer. Under certain conditions that are discussed later, phases can be merged. Refer to Data transfer summary section for details

Phase1 – MAC translation

FBUS protocol implements the phase1 of data transfer. Figure 31 illustrates ***FBUS*** cycle implementing first phase of data transfer protocol.

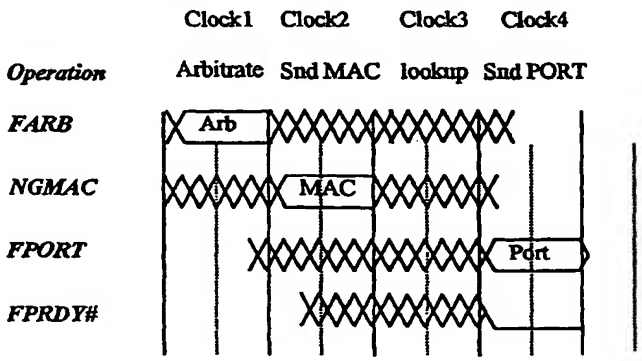


Figure 31 - MAC to PORT translation cycle

In clock1 *NGMAC* was arbitrated and ownership was granted for the next cycle. In clock2 MAC address is sent to FDB on the *NGMAC* bus. It is assumed that full clock is needed to send the MAC to FDB. Table lookup is performed in the next cycle (clock3), and in clock4 port address and its speed is returned on *FPORT* lines, qualified by *FPRDY#* signal.
Target queue port ID will be used by the receive queue to post data transfer request to transmit queue, it will be inserted to *TQID* field of the ***TREQ*** bus in transmit request phase.
The receive queue must examine the priority of the cell. If cell priority equals 15, then it should be routed to FSA port, regardless of port number replied by FDB. This is done in order to assure that FMP will not be discarded in case of the receive queue overflow.

Phase2 – Post transmit request

TREQ protocol implements second and third phase of data transfer – post transmit request and transfer data to transmit queue.

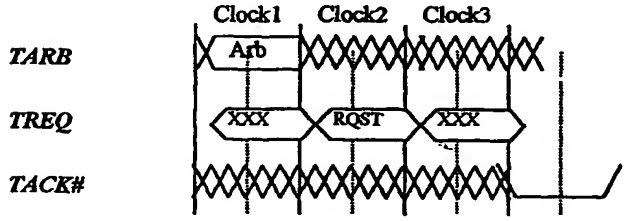


Figure 32 - acknowledged transmit request cycle

Figure 32 illustrates acknowledged transmit request phase. In clock1 ***TREQ*** bus is arbitrated with ***TARB*** bus, and transmit request is placed on ***TREQ*** bus in clock2. In clock4 ***TACK#*** signal is asserted by the transmit queue, indicating that request has been registered.

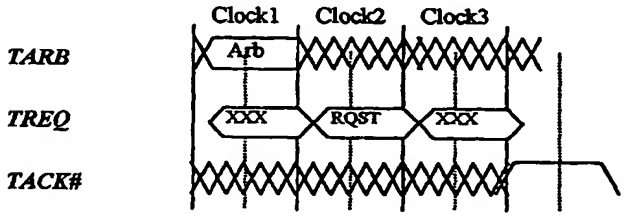


Figure 33 - denied transmit request cycle

Figure 33 illustrates denied transmit request phase. In clock1 ***TREQ*** bus is arbitrated with ***TARB*** bus, and transmit request is placed on ***TREQ*** bus in clock2. In clock4 ***TACK#*** signal negated by the transmit queue, indicating that request has not been registered, and receive queue needs to start transmit request phase over again.

Phase3 – Data transfer

Data transfer protocol concludes the transaction. Target queue requests data transfer from the receive queue using ***DiREQ*** bus. Receive queue transmits data to the target on ***DiBUS***.

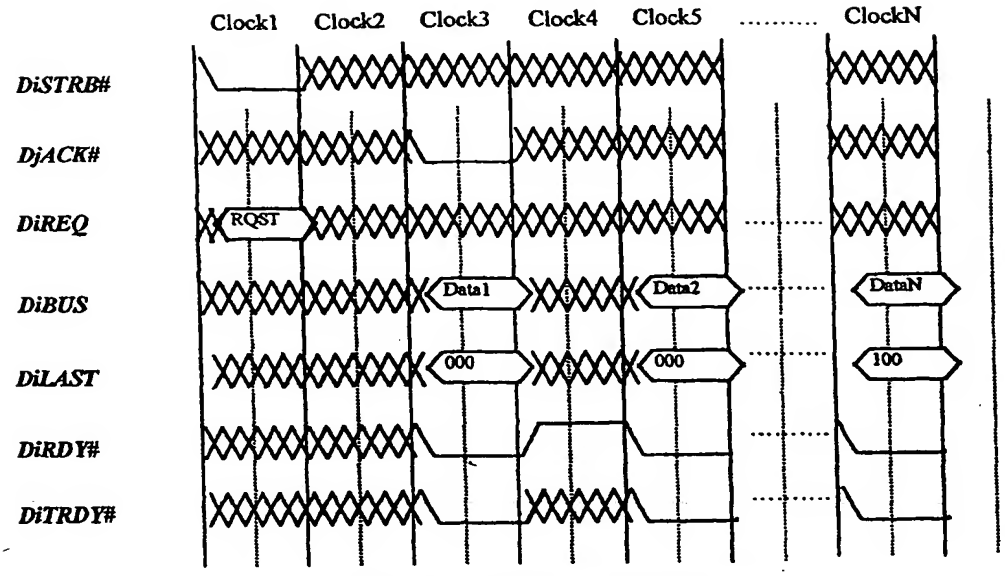


Figure 34 - successful data transfer phase

Figure 34 illustrates successful data transfer phase. In clock1 transmit queue places request for data transfer from the receive queue specified in *receive queue ID* field of ***DiREQ***, qualified by ***DiSTRB#***. All receive queues snoop ***DiREQ*** bus and one whose port number matches *RQID* field of ***DiREQ*** replies with ***DjACK#*** signal, assumes ownership of ***DiBUS*** and ***DiRDY#*** busses two clock after data request (clock3) and sends data to the transmit queue. The ownership of ***DiBUS*** and ***DiRDY#*** starts from the next cycle (clock3), and target queue should be ready to accept the data starting from clock3. Receive queue can reject the request from transmit queue by negating ***DACK#***. This will be done in case all output ports of the receive queue are busy (e.g. it is already transmitting to all directions). Once receive queue acknowledged the transmit queue request, it must send the first chunk of data no later than 5 clocks after transmit queue request was acknowledged.
Receive queue qualifies data transfer with ***DiRDY#*** signal. ***DiLAST*** signal is driven with '000 value in clock 3 and 5, indicating that it is not a last data transfer of the cell. ClockN is that last clock of cell transfer with all four ***DiBUS*** bytes valid, as indicated by the ***DiLAST*** value '100. Note ***DiTRDY#*** is asserted after every data transfer on ***DiBUS***, indicating that transmit queue accepted the data sent by receive queue.

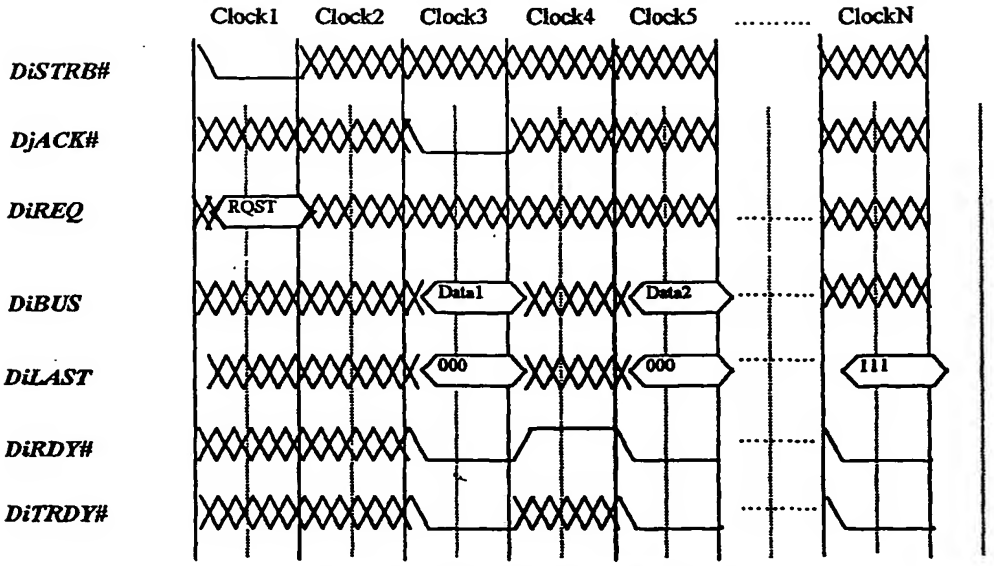


Figure 35 – data transfer completed with error

Figure 35 illustrates data transfer phase that terminates with error. In ClockN '111 value is placed on *DiLAST* signals by source queue, indicating that transmit of this cell should be terminated with Error Propagation Character (EP). Note that in this case value of *DiBUS* is ignored, and transmit queue appends EP character to the cell being transmitted. In case of error cell termination, the actual length transmitted by the receive queue can be below minimal cell length. Transmit queue must pad the short cell with dummy data while transmitting to assure that no illegal cell is generated by the MT101.

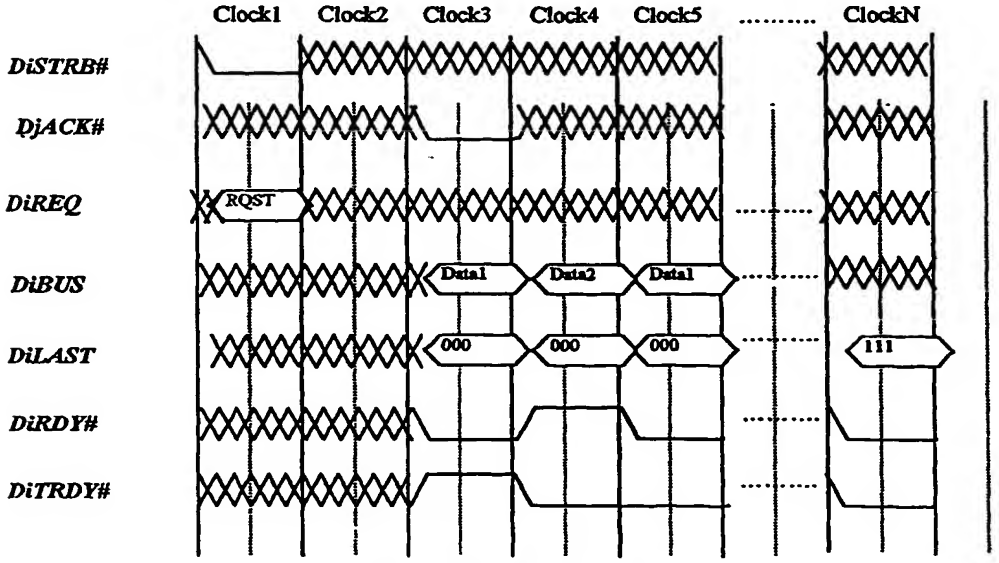


Figure 36 – re-try in data transfer

Figure 36 illustrates data transfer re-try. In clock3 valid data was placed by the receive queue on the bus (*DiRDY#* asserted), but it was not accepted by the transmit queue, as indicated by *DiTRDY#* being inactive in this cycle. Receive queue re-sends all data starting from the chunk that was negated by the target queue.
Figure 37 illustrates rejected data transfer request.

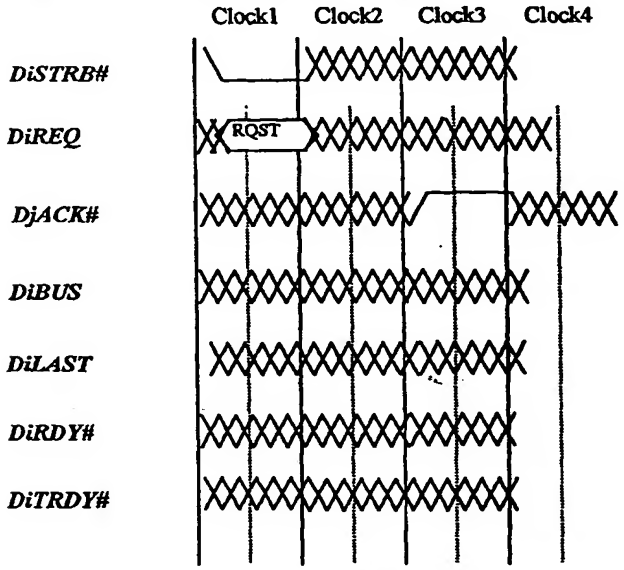


Figure 37 – rejected data transfer request

Data transfer request posted on *DiREQ* bus in clock1. In clock3 *DjACK#* was negated by the receive queue, which indicates to transmit queue that data transfer cannot start within a committed time limit, and therefore transmit queue needs to request data transfer again later. Note, that once data transfer request was negated by the receive queue, it means it did not assume responsibility to drive *DBUS* lines (*DiBUS, DiRDY#*) – e.g. receive queue that owned these lines must keep driving them in clock4 to avoid leaving them floating. Refer to Bus Drive Conditions summary **Error! Reference source not found.**.

Data transfer summary

This section will show full cycle of data transfer – from the clock MAC is available on the receive queue till first cycle of data transfer.

Three-phase data transfer

This section shows full cycle of data transfer that goes explicitly through all three phases:
The first event of the data transfer is arbitration for *NGMAC* bus. This can be done in the same cycle MAC being extracted form the header.
Figure 38 illustrates an explicit-phase inter-unit data transfer.

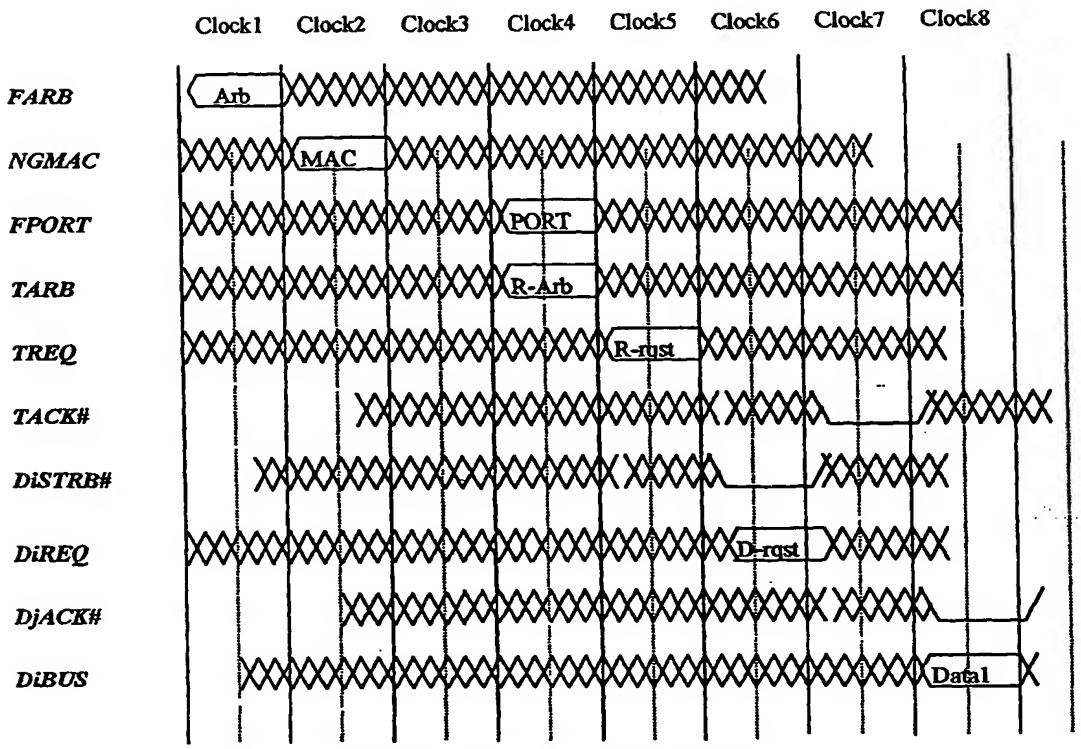


Figure 38 – 3-stage inter-unit data transfer

Inter-unit data transfer period starts in clock1, where receive queue arbitrates the ***NGMAC***, which is granted for the next cycle. In clock2 MAC address is sent to FDB, in clock3 FDB lookup is performed and in clock4 PORT value received from FDB, which concludes the first phase of data transfer. This phase can be shortened if receive port contains FDB lookup table (or some sort of FDB caching), which saves flight time on the busses.

Second phase of data transfer starts after first phase was concluded. With limited pipelining, second stage can start not earlier than in the clock PORT was returned to the requesting unit. The first event of the second stage is ***TREQ*** arbitration by receive queue (clock4). ***TREQ*** is granted for the next cycle, and data transfer request is placed on ***TREQ*** bus in clock5. In clock6 transmit request acknowledged (***TACK#*** asserted), which concludes second phase of data transfer.

Third stage of data transfer starts after second phase was completed. With limited pipelining, third stage can start not earlier than the clock transmit queue acknowledges the transmit request. The third stage starts when transmit and places data transfer request on ***DiREQ*** lines in clock6. In clock8 the first chunk of data can be driven by receive queue, qualified by ***DiRDY#*** and ***DiTRDY#*** (not shown on the figure).

Two-phase data transfer

Under certain conditions different phases of data transfer can be merged. If target queue is idle, the data transfer can start already in phase2, hereby merging phase2 and phase3 of data transfer.

Each transmit queue indicates that it is empty by asserting ***TQiIDLE#*** signal. Each receive queue observes all ***TQiIDLE#*** signals, and if data is targeted to idle transmit queue, data transfer can start in the same clock transmit request is placed on ***TREQ*** lines (clock5), reducing latency by 3 cycles.

Figure 39 illustrates the case where phase2 and phase3 of data transfer are merged.

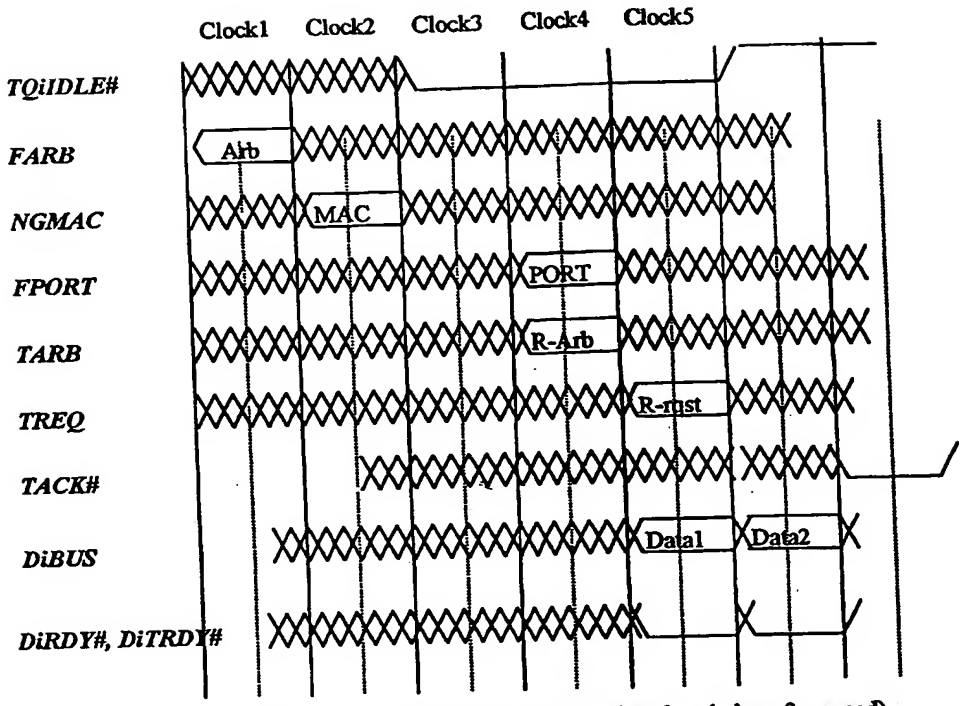


Figure 39 – two-phase data transfer (phase2 and phase 3 merged)

In clocks 1-3 MAC translation is performed as described above. While arbitrating for ***TREQ***, receive queue observes that merge conditions for phase2 and 3 met (refer to Protocol events section), and thus it places first chunk of data on respective ***DiBUS***, qualified with ***DiRDY#*** in the same clock is places the transmit request on ***TREQ*** bus. To simplify the observer's task, Receive queue indicates that it is phase-merge transmit request in ***TREQ.CMD*** field of the. Target queue asserts ***DiTRDY#***, acknowledging the data receive. Only if phase2 and 3 conditions are met, receive queue is allowed to drive data on the bus in the clock ***TREQ*** is placed. Note that if merge condition occurred, transmit queue will not place request on ***DiREQ*** for this data transfer anymore.

NGIO to NGIO latency can be improved further by implementing FDB lookup table in each receive queue or by caching most frequently used entries. See PCI to NGIO transfer discussion for details. This option will be evaluated at the later stage.

Note that receive queue speed is slower than transmit queue, receive queue needs to pay extra caution while merging phases. It can be done in two ways:

1. After receiving destination port information, nullify (force NOP) ***TREQ***. Don't attempt to arbitrate until enough data is buffered in the queue. This is preferred option from performance standpoint.
2. Delay arbitration for ***TREQ*** by one cycle and start arbitration only after enough data is buffered.

The choice between the two options is implementation-dependent.

If FDB is cached on the receive ports, phase1 of data transfer protocol can be performed internally on the queue, now showing up on the ***FBUS***. In this case data transfer latency will shorten by two cycles. MAC address of PCI will always be cached on every port, so transfers to PCI will take three cycles.

Performance summary

Table 13 below summarizes best case latency for data transfers.

Transfer	clocks
NGIO to NGIO	5
NGIO to PCI	3
PCI to NGIO	3

Table 13 - minimum data transfer latency

Fabric Management data transfers

The Fabric Service Agent (FSA) appears as NGIO port to the internal protocol with port address 9 (nine) – the largest port number. All accesses to port9 will access FSA.
FSA will contain the Fabric Management Packets Queue (FMPQ), and data will be transferred to the queue on *D9BUS* using data transfer protocol. If priority in the cell received by NGIO port is 15, it is FMP cell, and should be routed to the FMPQ.
Transfers to FMPQ are similar to transfer to any other port. Data is queued in the receive queue, request is posted to FMPQ (phase2). Eventually FMPQ will request cell transfer, and data will be transferred to FMPQ.
If FMPQ is empty, FSA will assert corresponding *TQiIDLE*, and newly arrived cell can be transferred from the NGIO port to FSA even if its data array is full. Each receive queue should preserve a bus driver it its data array is full and there is no pending FMP in the array. If there is no room in receive queue to place FMP arrived and FSA has FMP in process ('phase2 and phase3 merge' condition is not met), the FMP should be dropped by the receive queue.

Configuration registers access

Configuration registers are accessed using *CBUS* bus bundle, which contains of *CRBUS, CADS#, CRW#, CRSRC* and *CRDY#* signals. All configuration reads/writes are initiated by FSA, S-EPROM, PCI or 8-bit CPU interface unit and each unit responds to its registers' access per address specified in table yyy below. Configuration register (CR) address is driven on *CRBUS* in the address phase of the cycle, qualified with *CADS#*. Read/Write# and request source indication is driven with address on *CRW#* and *CRSRC* lines respectively. Each configuration register can be accessed from PCI, NGIO, and CPU. While being accessed from its 'natural' source (e.g. PCI configuration from PCI port), unit holding the register must obey access rules as specified (e.g. protect read-only registers from being written etc). While being accessed from 'other' source (e.g. PCI configuration registers from CPU), all register should behave as regular data storage – e.g. all bits are written on write operation and read on read operation.
CRBUS is not pipelined, there can be no more than one active cycle at a time, which assures no contention on the bus.
All registers are 32-bit registers, e.g. each *CRBUS* operation consists of one-clock address phase and onetwo-clock data phases. Figure 40 illustrates CR read and CR write operation.

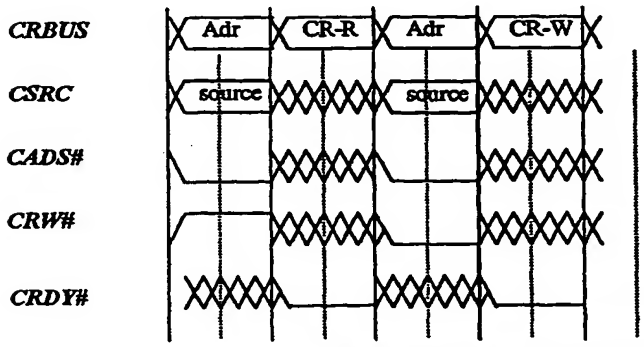


Figure 40 – CR read and write operation.

Configuration registers access can be initiated by CPU, S-EPROM, from PCI or from FMP.
FMP-originated accesses are performed by FSA, which breaks FMP into series of *CBUS* cycles per FMP

received. FSA contains CBUS arbiter, implementing '*HOLD/HLDA*' arbitration protocol. *HOLDi* used to force unit out of the *CBUS*, *HLDAi* acknowledges bus release. Figure 41 illustrates *CBUS* arbitration.

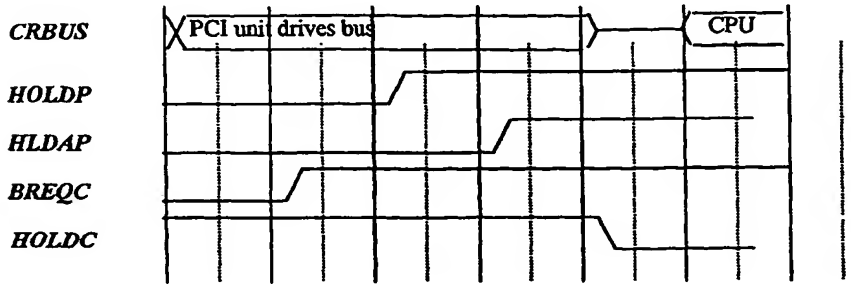


Figure 41 – CBUS arbitration

The figure shows example of CBUS arbitration between PCI and CPU units. At the beginning PCI owns *CBUS* and drives it. CPU units asserts *BREQC* signal, indicating that it needs *CBUS* to access the registers. Next clock arbiter asserts *HOLDP*, forcing PCI unit off the bus, and PCI unit acknowledges with *HLDAP*. Clock after it asserts *HLDAP*, it quits driving the bus. Arbiter clears *HOLDC*, and from now on *CBUS* is driven by the CPU unit

Cells squash (discard)

Due to various conditions, cells can be discarded by the MT101. Cells can be discard condition can be triggered in receive queue (invalid port etc) and in transmit queue (lifetime expired). Cells are discarded due to following conditions:

1. Invalid (garbage) target port. This condition is received from FDB at MAC translation phase.
2. Cell lifetime expiration – cell lifetime exceeded.
3. Error encountered in cell CRC – this will be done (if at all) in the receive queue.
4. Another conditions that I cannot think about now

The cell can be discarded by receive queue any time before transmit queue requested data from receive queue (e.g. any time before phase3 of data transfer started). If cell discard condition occurred before transmit request posted, the cell is discarded without any external notification. If cell discard condition occurred after transmit request posted to transmit queue, receive queue needs to notify transmit queue to cancel the transmit request. This is done by posting transmit request for the same cell for a second time to the same transmit queue with 'discard' opcode on *TREQ.CMD* lines. Transmit queue acknowledges removal of transmit request with *TACK#* signal. If cell squash request posted after data transfer started, transmit queue will deny this request (by negating *TACK#*), and receive queue will send the data to transmit queue.

If cell discard condition occurred after third phase of data transfer started (data transfer request acknowledged by the receive queue), receive queue can either cut the transmission by terminating data transfer with *ERROR* delimiter (forcing *DiLAST* signals to '111) or complete the transmission.

If cell discard condition occurred in transmit queue (life timeout), transmit queue notifies respective receive queue on *DREQ.CMD* field that specific cell needs to be discarded. This request implies implicit acknowledge from the receive queue (e.g. receive queue cannot reject or postpone such a request).

Transmit queue has a timer (counter) for each priority queue. When new entry reaches the top of the queue, the counter is loaded with LifeTime value of PortInfo COD field. On transition from '1 to '0, the discard event occurs and transmit queue posts discard command to the respective receive queue, removes killed entry from the head of the queue, places new entry to the queue and re-initiates the lifetime counter.

Data transfer responsibility

Once MT101/102 assume responsibility of the data transfer from PCI bus, it is guaranteed that data transfer will be completed within (programmable) period; else error will be signaled to the SW. In order to assure

data transfer completion, NGIO channels with PCI end-points must be connected/acknowledged channels. The data transfer ownership of MT101/102 is implemented through rules defined bel w:

1. All channels with PCI at the end point must be connected/acknowledged channels.
2. On memory reads PCI unit transfer the cycle to delayed read. PCI unit logs the read request (address, CMD, byte enables etc.) and generated RDMA-read NGIO cell. Subsequent retries for the same cycle by the host will be delayed by PCI unit until RDMA-read response with data arrival, when data will be buffered and sent to the host after subsequent retry.
3. I/O reads treated same way as memory reads.
4. Non-posted writes and I/O writes treated same way as memory reads. In addition to address and CMD, PCI unit also logs data written, and generates RDMA write NGIO cell, specifying that it is posted (I/O) write. After acknowledge received, PCI unit will compare data of re-try cycle, and if address, CMD and data matches original cycle, PCI unit returns TRDY# and completes cycle.
5. Posted writes generate RDMA-write NGIO cell, and returns TRDY# immediately to the host. The NGIO cell ID (MAC, PSN etc.) is kept alive in PCI unit until RDMA-write is acknowledged.

If acknowledge for the cell does not arrive within specified period (refer to configuration section), the event is logged, and can further generate interrupt to the Master Fabric Manager.

Global signals and events definition

Protocol events

Event	Condition
Phase2 and Phase3 merge	In the clock of arbitration for *TREQ*: 1. *TQiIDLE#* is asserted AND 2. no valid *TREQ* for the target port is snooped on *TREQ* in that clock
TQiIDLE# assert	Clock before transmit port can unconditionally receive new cell data (the earliest)
TQiIDLE# negate	Clock after first data chunk was driven to the port.

Table 14- Global events summary

Global signals' summary

Table below summarizes global signals and busses of the device. Timing of each bus is specified as

1. Early (driven from the latch, available early in the cycle),
2. Medium (goes through several logic gates, available in the middle of the cycle)
3. Late (goes through significant logic before driven out, available in the late part of the cycle, should be sampled without excess logic load).

Bus name	Description	Tim	#	wires
FARB[8:0]	Arbitration bus, used by all ports to arbitrate for *NGMAC* bus and FDB port for MAC to port translation. The bus arbitration is done as described in Arbitration Protocol section. Bits 8:0 are assigned to the remaining ports and deploy cyclic priority arbitration.	E	1	9
NGMAC[15:0]	This bus is used to send MAC address for translation.	E	1	16
FPORT[6:0]	Used to send the target port number and its speed from FDB to receive queue. If *FPORT6* bit of *FPORT* is clear, *FPORT[5:0]* contains valid port address and speed. If *FPORT6* is set, *FPORT[5:0]* encodes special cases: '100000 – discard cell '1xxxxx – reserved For more details please refer to FDB unit spec. *Port – FPORT[3:0]* – port address *Speed – FPORT[5:4]* – port speed	M	1	7
FPRDY#	An *FPORT* qualifier. *FPORT* lines are valid only if *FPRDY#* is asserted.	E	1	1
TARB[9:0]	Arbitration bus, used by receive queues to arbitrate *TREQ* bus. The bus arbitration is done as described in Arbitration Protocol section. *TARB9* (the most significant bit, corresponding to highest priority) is always assigned to the PCI port, which assures highest priority for PCI. Bits 8:0 are assigned to the remaining ports and deploy cyclic priority arbitration.	M	1	10

Bus name	Descripti n	Tim	#	wires
TREQ[36:0]	Request bus, used in phase2 of data transfer to post transmit request. *TREQ* bus has following fields: *CMD – TREQ[2:0]* decodes command to be executed by ports. Following commands are supported: 000 – NOP. All fields of TREQ bus should be ignored. No bus drive responsibility changed. 001 – Discard cell. Used by receive queue to discard transmit request that was posted already to transmit queue. 010 – transmit request. Driven by receive queue while posting request for transmit queue in second phase of data transfer protocol 011 – phase2 and phase3 merge. 1xx – Reserved. *TPID – TREQ[6:3]* contains port number of the queue this request is targeted for. All transmit queues must snoop *TREQ* bus one clock after it was arbitrated and if *TPID* matches port number mapped to the queue, it must respond to the request (acknowledge/deny). *RQID – TREQ[10:7]* contains Receive Queue ID (port number). *Priority – TREQ[14:11]* contains cell priority after re-map from NGIO to MT101 priority queues.. Valid values are 0,1,2,3 and 15 *CellID – TREQ[19:15]* contains ID of the cell in the receive queue. *DataAdr – TREQ[28:20]* contains address of this cell in receive queue data array *Opcode – TREQ[36:29]* contains opcode field from the NGIO cell.	E	1	37
TACK#	Indicates that request posted on *TREQ* in one before previous cycle was accepted.	E	1	1
DiSTRB	*DiREQ* bus strobe. If *DiSTRB#* asserted, valid data request is posted on *DiREQ* bus.	E	1	10
DiREQ[19:0]	Request bus, used in and phase3 of data transfer. Each transmit queue has *DiREQ* bus associated with it. Each receive queue observes all *DiREQ* busses *DiREQ* bus has following fields: *CMD* – DiREQ[1:0] contains to command being sent to the receive queue: '00 – data transfer '01 – discard cell '1x - reserved *RQID – DiREQ[5:2]* contains Receive Queue ID (number). All receive queues must snoop *DiREQ* bus. If *RQID* field matches the receive port number, then it is a request for data transfer from receive queue to transmit queue. *CellID – DiREQ[10:6]* contains ID of the cell in the receive queue. *DataAdr – DiREQ[19:11]* contains address of the cell in receive queue data array	E	10	200

Bus name	Description	Tim	#	wires
DjACK#	Indicates that request posted on *DiREQ* in one before previous cycle was accepted. Each receive queue has its own *DjACK#* signal. Transmit queue that requests data from the receive queue should observe respective *DjACK#* clock after data request was posted on *DiREQ* bus.	E	1	10
TQiIDLE#	Indicates that target queue *i* is idle and can receive cycle unconditionally. Exact timing of this signal is specified in Key Events section.	E	10	10
DiBUS[15:0]	Data bus, used to transfer data from receive queue of any device port to transmit queue of port *i* (the port associated with this bus).	M	10	160
DiLAST[2:0]	Indicates status of the current data transfer. Using the following encoding: 000 – both bytes of *DiBUS* are valid and more data corresponding to this cell is expected to be transferred by receive queue that currently drives the *DiBUS* 001 – last transfer of the cell with one valid byte. 010 – last transfer of the cell with two valid bytes. 111 – last transfer of the cell with error. The cell transmit should be terminated with Error Propagation Character (EP), data driven on *DiBUS* is invalid	E	10	30
DiRDY#	Signal indicating that corresponding *DiBUS* and *DiLAST* have a valid data driven by receive queue	E	10	10
DiTRDY#	Signal indicating that transmit queue has accepted (latched) *DiBUS* and *DiLAST* values from the respective busses. If *DiTRDY#* is inactive, receive queue must re-transmit the data starting from the chunk that was not accepted by the target queue. To avoid deadlocks (*DiRDY#* and *DiTRDY#* oscillating), once *DiTRDY#* is asserted, it cannot be cleared before valid data chunk arrived (similar to PCI bus *TRDY#* rules)	E	10	10
RQjSTAT[7:0]	Request queue status, provides auxiliary information that can be used by transmit queue for arbitration decision. Usage of this bus is not mandatory, it can help to improve throughput and system utilization in high loads. *RQjSTAT* bus has following fields: *Hprio – RQjSTAT[1:0].* This field indicated highest priority request pending in the request queue. *Fchan – RQjSTAT[3:2].* This field indicates number of channels free for data transfer in receive queue. If *RQjSTAT.Fchan* = '00, it means that all channels are busy with data transfer and request for data transfer will be denied (refer to Phase3 – Data transfer section) *FC – RQjSTAT[7:4].* This field indicates proximity of different priority to flow control.	E	10	80
CRBUS[31:0]	Bus used to read/write control and configuration registers of the device. Refer to Initialization and configuration section for protocol	E	1	32
CADS#	Indicates that valid configuration register address is placed on *CRBUS*. Initiates CR access		1	1
CRW#	Indicates whether CR access is read or write.		1	1
CRDY#	Indicates that valid CR data is placed on *CRBUS* for CR reads. Indicates that data placed n *CRBUS* for writes has been sampled by the target.		1	1

Bus name	Description	Tim	#	wires
CRSRC[2:0]	Indicates source of register access: 000 – access initiated from PCI port 001 – access initiated from NGIO port 010 – access initiated from 8-bit CPU port 011 – access initiated from Serial EPROM 1xx – Reserved.		1	3
HOLDC, HLDAC, BREQC	*HOLD/HLDA/BREQ* signals for CPU unit, used for *CBUS* arbitration.		1	2
HOLDE, HLDAE, BREQE	*HOLD/HLDA/BREQ* signals for EPROM unit, used for *CBUS* arbitration.		1	2
HOLDP, HLDAP, BREQP	*HOLD/HLDA/BREQ* signals for PCI unit, used for *CBUS* arbitration.		1	2
INIT	Initialization process being performed	E	1	1
RESET	Global HW reset	E	1	1
				647

Table 15 - global signals summary

NGIO port external definition and requirements

NGIO port consists of the following basic blocks:

1. Receive queue – responsible to receive NGIO cell, inquire FDB to translate MAC to PORT address, notify appropriate transmit port and deliver data upon transmit port request. Receive queue simultaneously transmit upto 4 cells to 4 different transmit queues.
2. Transmit queue – responsible to accept transmit requests from receive queues, arbitrate the link and request data to be transferred from the receive queue.
3. Link maintenance machines – responsible to maintain NGIO link, generate delimiters, identify link failure.

Link maintenance machines – details.

Link check machines are responsible to maintain link, identify link existence (good link), synthesize and transmit adequate control characters.
After initialization sequence is completed (*INIT_DONE* signal assertedcleared), the Link Machine establishes channel connection at speed specified in the port configuration register.
Receive Link machine identifies delimiters and notifies the Receive queue when new cell arrives. Transmit link machine generates delimiters between the cells.
Link machine checks the link status as specified in NGIO Link document. In case of link failure, link machine sets LinkDown bit in the port status register. This register can be read by SW (through FMPs or from CPU side) and by HW (FSA, transmit queue).

In case of link disconnect, all flow control from this link should be removed. Transmit queue will squash all data sent to it, hereby acting as /dev/null – this is in order to flush all cells pending transmission to that port.

Receive queue - details

NGIO receive queue is responsible for

1. accept NGIO cells
2. initiate FDB inquire to translate MAC address to port number
3. request data transfer from target port. Requests should be sent to target port in order of their arrival within same priority to avoid illegal out-of-order transmission. Higher priority cells should be scheduled for transfer before low-priority ones.
4. deliver cell data to the target port upon request
5. keep track of cell's age and squash expired cells
6. issue flow control messages to avoid queue overflow by inbound traffic
7. Check cells for errors (CRC) and inject EP as necessary (not a MUST per switch spec, but good feature to debug the network).
8. Squash cells that arrived with EP delimiter (configuration).

Receive queue should fully implement data transfer protocol (all phases).
Receive queue consists of two major blocks

1. Data array. Data array stores cells that were received by the transmit queue. Array size is 292x7 bytes, e.g. it can contain upto 7 NGIO cells of maximum length.
2. Cell pointers. This is a register file of 16 entries, which contains pointers to cells in the array. The pointers contain all cell information needed to post request, make decision about cells' squash etc.

Receive queue should generate flow control, which can have two distinct sources:

1. The data array runs out of space. Flow control will be issued based on space left, the watermarks are programmable, A1...A8 specifies number of left empty in the array. A1<=A2<=...<=A8. The values of A1...A8 are programmable through the receive queue control register.

Number of empty entries	Flow control
A1	XN7
A2	XN6
A3	XN5

Number of empty entries	Flow control
A4	XN4
A5	XN3
A6	XN2
A7	XN1
A8	XN0

Table 16 - Flow control conditions - data array

2. Receive queue runs out of cell pointers. In this case flow control should be issued according to the rules summarized in the Table 17. N1<=N2<=...<=N8. N1...N8 values are programmable in the receive port control register

Number of empty entries	Flow control
N1	XN7
N2	XN6
N3	XN5
N4	XN4
N5	XN3
N6	XN2
N7	XN1
N8	XN0

Table 17 Flow control conditions - pointers

Transmit queue - details

NGIO transmit queue is responsible for:
1. accept data transfer request from receive queue of (other) NGIO port
2. acknowledge the acceptance of data transfer request
3. resolve priority of all pending transmit requests
4. inquire data for transmission from the corresponding receive queue

Transmit queue should log requests from the receive queues and arbitrate the outbound link. Transmit queue should collect enough information to make a right decision during second phase of data transfer (logging requests from the receive queues). Transmit queue will take into consideration following constrains while arbitrating the link (in priority order):
1. Priority of outstanding requests. Higher priority request should be transmitted first
2. Load on the receive queue. Requests from receive queue with higher load should be served before requests from queues with lower load
3. Non-starving of low-priority requests, implementing LiveLock register (refer to LiveLock section for details)

More information can be collected by the transmit queue (like cell length etc.). It is not clear at this point whether additional information is necessary.
In case of link disconnect, all flow control from this link should be removed. Transmit queue will squash all data sent to it, hereby acting as /dev/null – this is in order to flush all cells pending transmission to that port.

PCI port external definition and requirements

MT101 PCI port supports up to 66MhZ PCI bus. MT101 supports 32 and 64-bit PCI with full 64-bit address space support.
Once PCI slave assumes responsibility on the cycle, it is responsible to assure its completion, else error (interrupt) will be issued. Hence, all PCI-originated channels are connected/acknowledged channels – no exceptions.
PCI port consists of the following basic blocks:

1. NGIO port associated with PCI port, implementing interface of PCI to NGIO world.
2. PCI bus master block – responsible to translate requests from NGIO network to PCI and return the response back to NGIO world
3. PCI slave block – responsible to accept PCI requests, translate them to NGIO world by issuing appropriate NGIO cell, accept response from NGIO network and translate it back to PCI world.

NGIO port – details

NGIO port is designed in such a way that it can interface to PCI block, so it can be used in PCI unit almost without changes. PCI unit interface to NGIO port similarly to the way NGIO link interface.
Since both master and slave of the PCI may have same MAC address and hence same port address, the transmit queue of NGIO port associated with PCI needs to take *TREQ.Opcode* field into consideration while responding to the *TREQ* request. If *TREQ.Opcode* contains response opcode, the request is targeted to the PCI slave. Otherwise it the cell is targeted to the PCI master unit.
In order to assure PCI cycles ordering is preserved[4], following rules must be followed[5]:

1. Cycles should be sent to NGIO fabric in same order they are issued on PCI bus.
2. Cycles received from NGIO fabric should appear on PCI bus in same order they received form the fabric.

PCI master and slave units will assure that requests are ordered before entering the common receive queue, see PCI master and PCI slave MAS for details.
The ordering of received cells will be assured by the common transmit queue, which will not accept new cell from NGIO fabric unless it is 'safe' to forward it to the PCI master or slave. Table 18 defines condition for transmit queue whether to accept incoming cell – depending on its opcode and status of the PCI master.
PCI master status indications:
WP – Write Pending. Write request accepted from NGIO fabric, but write cycle on the PCI bus has not been completed. Master can accept additional writes
WF – Write buffer full. PCI master write buffers are full, master cannot accept additional write requests
RF – read buffer full. PCI master cannot accept additional read requests.
Possible NGIO cells arrive to the PCI transmit queue:
SND – NGIO-send request (Opcode '0 through '101)
WR – RDMA-write request (opcode '110 through ''1011)
RD – RDMA-read request (opcode '1100)
RR – RDMA-read response (opcode '1101 through '10000)
ACK – Acknowledge response (opcode '10001)
Table 18 summarizes conditions when transmit queue will ask for a respective NGIO cell to be transferred to the PCI unit.

[4] The most challenging case here is to assure that rule 3 for 'transaction ordering and posting for bridges' is enforced (page 42 of PCI spec)
[5] The priority of the cell is not taken here into consideration for simplicity. NGIO fabric will take care about it and high-priority cells will pass over low-priority ones.

Cell\Status	WP	WF	RF	Otherwise
SND	Yes	No	Yes	Yes
WR	Yes	No	Yes	Yes
RD	No	No	No	Yes
RR	No	No	Yes	Yes
ACK	Yes	Yes	Yes	Yes

Table 18 - PCI transmit queue data request conditions

PCI master – details

PCI master is responsible to translate NGIO cells targeted to PCI to PCI cycles, issue the cycles to PCI, generate response cell and send it back to NGIO network.
Following requests (NGIO cells) should be served by the PCI master:

1. RDMA-read. Master should issue read on the PCI bus with length specified in the request, collect all data and construct RDMA-read response packet. If for any reason read could not be completed, master should send NACK as a response to the RDMA-read.
2. RDMA-write. Master should perform write operation of the PCI bus and send acknowledge cell back to the originator.
3. Other cycles (e.g. configuration) are treated similarly – each one is performed on the bus and acknowledge sent back to the originator.

The ACK/NACK payload for PCI master replies should follow NGIO Link spec, pp63-66

PCI slave – details

PCI slave is responsible to translate PCI cycles to NGIO cells and send them to the target on NGIO network. There are two ways to originate NGIO cell from the PCI:

1. Construct the cell explicitly in internal MT101 register and send to the fabric
2. Transform PCI cycle targeted to MT101 to NGIO cell.

The first option can be used for explicit (SW-visible) access of NGIO network (e.g. sending FMPs, initialization etc.), refer to SW generation of NGIO cells section for details.

FSA unit external definition

FSA unit (Funit) contains two key blocks – Fabric Service Agent (FSA) and Forward Data Base (FDB). FDB is responsible to translate MAC to port, including special cases handling (e.g. default port, squash port). FSA is responsible to accept all FMPs arrived to the device and either take appropriate action (if addressed) or forward FMP further. Note, that receive queues will forward all priority15 cells to FSA. FSA is also responsible to arbitrate CRBUS.

FDB

FDB contains a MAC to port translation table, which is mapped to MT101 control register space and is accessed through CRBUS.
FDB should implement the phase1 of data transfer protocol, delivering port number in a response to the MAC address. FDB also maintains the default port number (the one mapped to port number 255) and returns a physical port number if cell is routed to the default port. The port number is returned on lower 4 bits of ***FPORT*** bus in the phase1 of data transfer.
FDB also contains information about port speed. This information is returned on bits 4,5 of ***FPORT*** lines. It is used by receive queue to decide about cell buffering before its transfer to transmit queue. The port speed information encoding is defined in Table 1.
In case MAC address translation requires cell discard (e.g. routed to port 254 or invalid MAC), FDB should return value of '1000000 on ***FPORT*** lines, and receive queue will discard the cell.

FSA

FSA unit has several functions:
1. Respond to FMPs arrived to MT101
2. Generate FMPs due to events generated inside MT101 and wait for acknowledge.
3. Implement System Port as specified in SW generation of NGIO cells section.

Response to FMPs

FSA unit will accept all FMPs arrived to the MT101. It is mapped to internal port and should implement the internal data transfer protocol. FSA must have at least one buffer of full-size cell (292 bytes), where FMP will be placed upon arrival. FSA should decode the cell and act according to FMP content. The possible actions could be:
1. Forward cell to its destination – in case this FMP was not targeted to this device or its FSA
2. Execute read/writes to control registers of MT101 – in case this cell is FMPSet() or FMPGet(), addressed to this device. Construct – if needed – the reply FMP and send it to its destination. In case reply FMP cannot be constructed (COD requested is not supported in HW) – generate event (interrupt notification) and send it to Fabric Manager
3. Execute Direct Route protocol – upon arrival of Direct Route FMP.

Z-unit external definition and requirements

Z-unit is a unit that contains various miscellaneous functions. These functions do not necessarily form an overall bigger function (as other units), but rather grouped together for 'ease-of-management' purpose. Each sub-section of this chapter defines individual function of Z-unit

JTAG

JTAG unit is responsible to implement fully IEEE-compatible JTAG.

S-EPROM interface

Implements interface of MicroWire serial EPROM (spec is available in Data Sheets folder of th eOutlook Public Folders) and implements all features defined in Serial EPROM – initialization section
S-EPROM unit (Sunit) is responsible to interface with S-EPROM. On power-up, this unit asserts ***INIT*** signal and starts reading the contents of S-EPROM and loads all control registers of MT101/102. The data format in EPROM is formed as a pair of 16-bit address and 32-bit data. Address with value of 0xffff means that all values to be written to the registers are read. After last control register is loaded, S-EPROM interface unit will clear ***INIT*** signal, so device will continue the boot process.
Sunit enables to program S-EPROM through *ROMDATA* and *ROMSTAT* register. This is 32-bit register that can be accessed by SW from PCI, FMP or CPU interfaces.
ROMDATA register contains 16-bit address and 16-bit data to be written to the ROM. ROMSTAT register is used to control the S-EPROM write operation:
Bit0 – write enable. After this bit is set, Sunit writes contents of ROMDATA[31:16] to address specified in ROMDATA[15:0]. This bit is cleared by HW after write has been completed.
Bit1 – write in progress. If set, it means the previous write command did not complete, and writes to ROMDATA register are ignored.
Bit2 – read enable. After this bit is set, Sunit reads contents (2 bytes) from the address specified in ROMDATA[15:0] and places it to ROMDATA[31:16]. This bit is cleared by HW after read has been completed
Bit3 – read in progress. If set, t means that previous read command did not complete and reads from ROMDATA register will return undefined data. Writes to ROMDATA register will be ignored

CPU interface

CPU interface unit (Cunit) is responsible to interface with auxiliary CPU that can (optionally) be attached to the MT101/102. CPU can implement local network management, therefore Cunit should enable CPU to access all relevant resources in the network. In particular, it provides hooks for:

1. Construct NGIO cell and send it to the fabric
2. Receive NGIO cell from the fabric
3. Access all configuration register of the device.

The first two functions are similar to what is provided by the PCI unit (see PCI spec for details). Access to internal registers is done by implicit addressing of these registers from the CPU interface (e.g. read will be interpret as control register read; write will be interpret as control register write).

Appendix B – reference designs (common blocks)

Arbiter reference design

Arbiter reference design with mixed static and cyclic priorities is shown on Figure 42. This design must be instantiated in each unit that arbitrates on any given bus. In this implementation ARB lines that participate in cyclic priority are implemented as tristate bus. Alternative implementation is possible when ARB is a non-tristate bus and priority of each line is re-mapped every cycle.

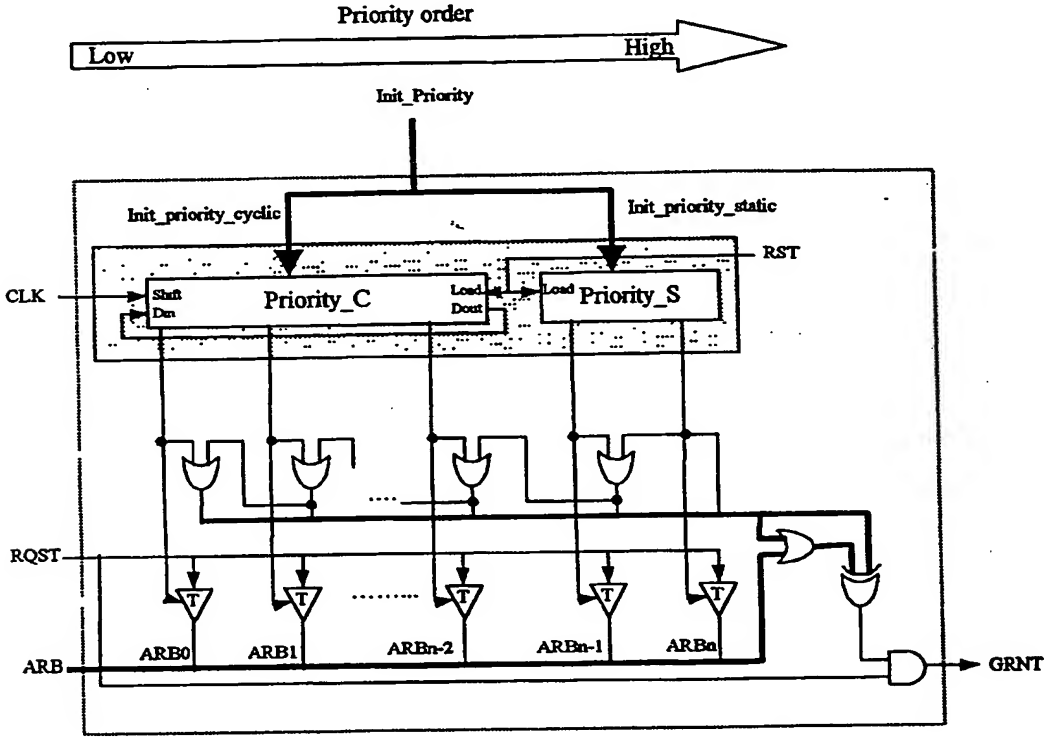


Figure 42 – Arbiter reference design

Priority register consists of two fields – *Priority_C* and *Priority_S*, which represent Cyclic and Static part of arbitration priority respectively. Initial value *Init_Priority* is loaded to priority register at initialization stage (reset). Only one bit can be set in this value. If set bit is loaded to *Priority_C* field of priority register, cyclic priority will be deployed for this unit. If set bit is loaded to *Priority_S* field of the priority register, this unit will be assigned static priority according to the set bit placement and will always have higher priority than units that assigned cyclic priority on this bus.
Note that if static part of priority is deployed on a bus, all priority registers must be of the same size. *Priority_C* and *Priority_S* fields must be of the same size in all priority registers.
Different units that use same ***ARB*** lines must be loaded with different *Init_Priority* value, and hereby each one will have unique priority at reset.
Priority_C part of the priority register is a rotate register. Each clock cycle *Priority_C* value is rotated by one bit, which results in priority change.
Every cycle every unit drives ***ARBi*** bit of the ***ARB*** bus, where *i* is a priority of this unit in that particular cycle. If unit has arbitration request pending (*RQST* input asserted), it will assert ***ARBi*** line. Otherwise ***ARBi*** line will be driven to inactive value.

Internally each unit generates *MASK*, extending its priority bit to the low end of the priority field. This mask is OR'ed with the *ARB*, and result is compared to *MASK*. If match occurred, the bus is granted to the unit (*GRNT* output asserted), and unit should drive bus it arbitrated for in th next clock cycle.

Appendix C – performance analysis

Latency analysis

The overall latency time between data receive and transmit is divided to three periods:
Period1 – time needed by the receive unit to get data from pins and place request for data transfer on internal bus
Period2 – time needed to arbitrate and deliver data over the internal bus network. Refer to the Data Transfer Summary section
Period3 – time needed by transmit queue to start data transfer on the pins from the clock it got the first data item of the cell.
For latency performance analysis it is assumed that there is only one transfer in the system between receive and target port.

Inter-unit protocol

Inter-unit protocol covers *Period2* of the data transfer. This period starts when MAC being extracted from the cell header.

PCI to NGIO latency

NGIO to PCI latency

NGIO to NGIO latency

Bandwidth analysis

Appendix D – MT101 resources' summary

MT101 pinout

Table 19 summarizes number of pins (signal and power) for MT101 device. It is assumed that a power pair (VCC/VSS) is needed for each 3 I/O pins.

Port	Signals	Freq. (Mhz)	I/O Voltage	Power (Vcc/Vss)	# of ports	Total pins (signals+power) x # of ports	Comments
PCI	95	66	3.3V drive 5V tolerant	16/16	1	127	
NGIO	22	133	??	7/7	8	288	Double-pump
S-EPROM	6	66 ??	??	2/2	1	10	
J-TAG	5	33 ??	??	2/2	1	9	
Fabric Mngr	0						
Core power	0			32/32		64	Validate with Shai. Based on Ronni's estimations
8-bit CPU	10						
RESET+misc	5						
Total						64	

Table 19 - MT101 external pins summary

MT101 arrays enumeration

Table 20 summarizes main memory arrays in MT101 device. All sizes are given in bytes. Numbers can (and will) vary while tuning architecture. The table is not guaranteed to reflect latest uArch POR.

Unit/block	Size (bytes)	# of units	Total	comments
NGIO transmit	32	10	320	
NGIO receive	292x8	10	23360	4 cells in each of the four prio que + FBQ.
FSA registers and tables	1024	1	1024	
FDB (MAC/port translation table)	32	1	32	8 ports with 8 MAC each. 3 bytes per entry
PCI outbound queue (write posting etc)	292x4	1	1168	Assumes 4 outstanding PCI cycles. Seems too low
PCI inbound queue	292	1	292	
PCI prefetch buffer	256 + 20	1	276	Data + tags
PCI cell templates	20x4	1	80	Templates for various NGIO cells
PCI configuration stuff	512 (?)	1	512	Configuration registers
Other configuration and mode registers	32K		32768	
Total			59832	

Table 20 - MT101 arrays summary

Arrays details

NGIO receive port arrays:

4 arrays of 16x256 – data (will be enough for 7 cells. Maybe will have to grow to 16x512 each.
One array of 'next pointer' – 8x256. If data will grow to 16x512, this one will have to grow to 9x512.

NGIO transmit port

MT101 micro-architecture specification

Rev0.5

Michael Kagan
Freddy Gabay
Mellanox Technologies

MT101 and MT102 overview

MT101 architecture is a baseline architecture that is implemented in multiple products, first being MT101 and MT102. MT101 block diagram is shown on Figure 1, and MT102 block diagram is shown on Figure 2. As could be noticed from these figures, MT102 is a subset of MT101 component. MT101 internal architecture design is targeted to simplify MT102 design. The inter-unit protocols have no notion about number of units and their nature. Global chip resources are not limited to any fixed number of NGIO or PCI ports. This document will describe the MT101 architecture.

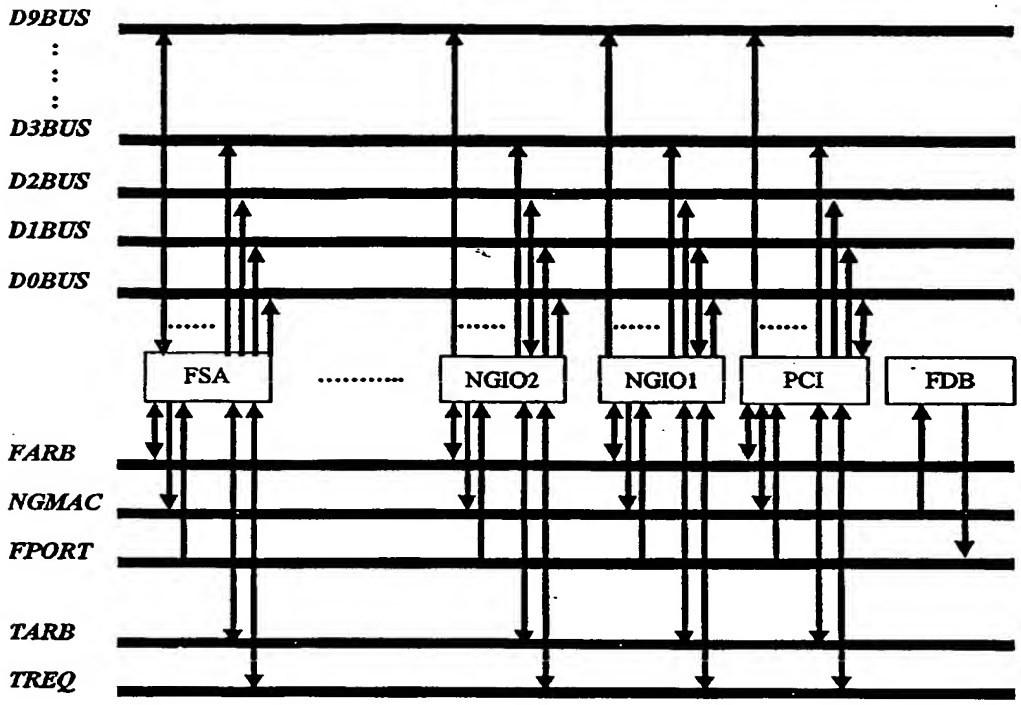


Figure 1 - MT101 block diagram

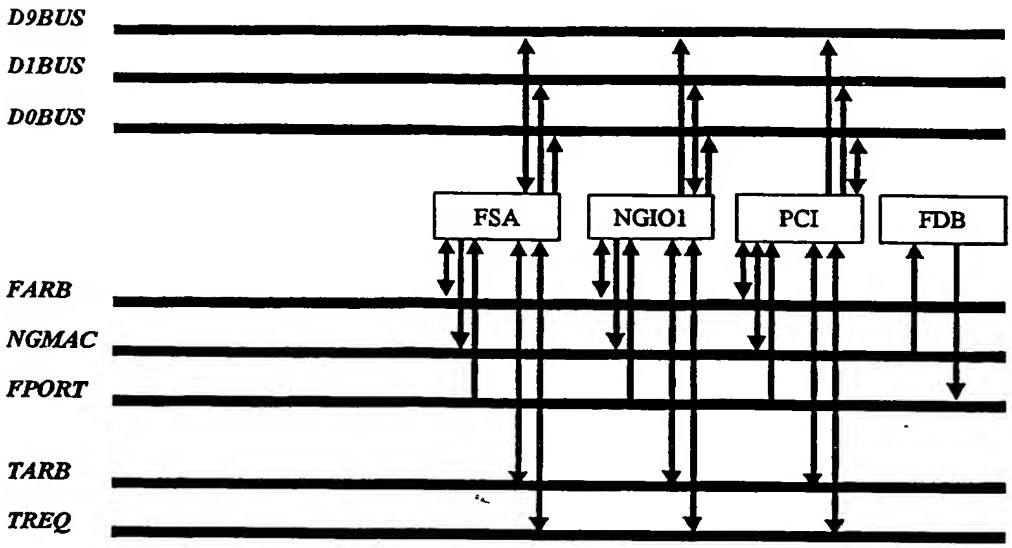


Figure 2 - MT102 block diagram

There are three groups of the busses:

1. *FBUS* group (including *FARB*, *FPORT* and *NGMAC*). This group implements phase1 of the transaction protocol.
2. *TRQ* group (including *TARB* and *TREQ*). This group implements phase2 of the transaction protocol
3. *DBUS* group, which includes data busses. This group implements phase3 of the transaction protocol.

Block description

NGIO unit is NGIO port, implementing NGIO receive and transmit queues. Detailed description of NGIO unit is available in .

FSA unit is a Fabric Service Agent unit, responsible for perform all fabric management functions of MT101 device. Detailed description of FSA unit is available in chapter.

PCI unit is PCI port, responsible to accept PCI cycles and route them to NGIO network. It is also responsible to transfer NGIO network requests for PCI resources to PCI bus. Detailed description of PCI unit is available in .

Each unit is has a data bus associated with it, which is used to send data to the unit for transmission. Multiple ports can be mapped to same unit.

Data transactions

Arbitration protocol overview

This section describes arbitration protocol used by all busses. Distributed arbitration will be deployed in on all MT101 busses, which avoids long round-trip paths and enables easy scalability of the architecture. General connection for a bus to be arbitrated is shown at the Figure 3 below. The scheme assumes N devices that connected to the same bus ***BUS*** and they use ***ARB[n-1:0]*** signals for arbitration.

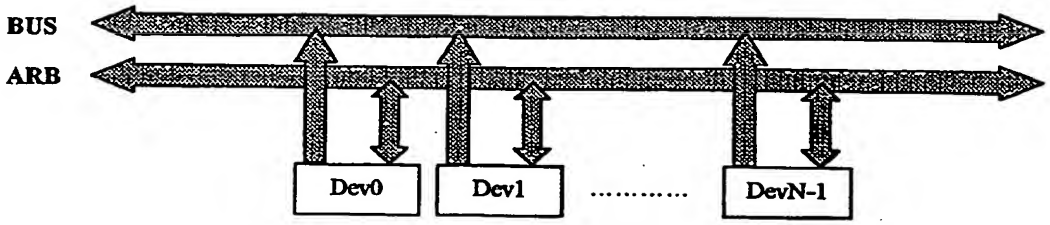


Figure 3 - bus arbitration - general case

Each device *m* has an ***ARBm*** line associated with it. This is the line it drives to active state if it wants to acquire ownership on BUS. The position of *m* bit in ***ARB[n-1:0]*** signal identifies priority of this device in arbitration. The higher *m* is, the higher is priority of the device. Protocol assures that if two devices – *m* and *n* (*m*>*n*) requested ownership of the bus at the same clock, device *m* will acquire ownership, and device *n* will give up, as priority *n* is lower than *m*.

At the beginning of arbitration cycle, every device puts its request for ***BUS*** on respective ***ARBi*** line (where *i* is priority of the device at that point). At the end of this cycle each device observes entire ***ARB*** signals. If no higher priority request was placed on ***ARB*** signals, it means the arbitrating device granted ownership of ***BUS*** and can drive it next cycle. If devices notices request of higher priority than is own placed on ARB signals, it means arbitration of is failed and ***BUS*** ownership was not granted. The device can arbitrate again in the next cycle. Arbitration protocol is fully pipelined. Figure 5 illustrates arbitration between 3 devices.

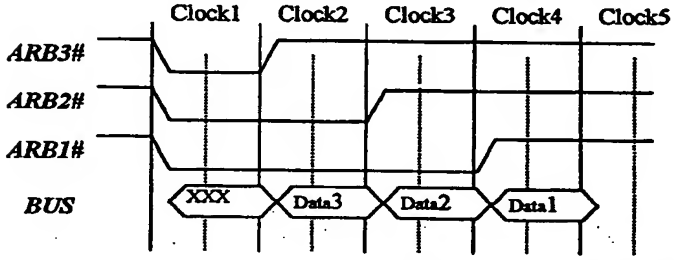


Figure 4 - arbitration example

In clock1 all 3 devices (priority 1,2 and 3) place their request for ***BUS*** arbitration. At the end of clock1 by observing ***ARB*** lines, device1 and device2 noticed that higher priority device (device3) requested bus in the same clock (***ARB3#*** asserted), and therefore their arbitration failed. Device3 assumes ownership of the ***BUS*** for the next cycle.

In clock2 device3 drives his data on ***BUS*** lines, and devices 1 and 2 arbitrate again on the ***BUS***. Device1 notices that higher priority device (device2) requests the bus and gives up. Device2 does not see any higher priority arbitration requests, and therefore it is granted BUS ownership for the next cycle.

In clock3 device2 drives his data on ***BUS*** lines, and device1 arbitrates again. This time no higher priority requests posted, therefore device1 is granted the bus and drives its data on clock4

Static priority

Using arbitration protocol described above, static priority between devices can be achieved by static assignment of *ARB* lines for each device. Thus, if device *m* always drive *ARBm#* line in arbitration cycle, and device *n* drives *ARBn#* (*m>n*), this means device m will always have higher arbitration priority than device *n*.

Cyclic priority

Static priority has a drawback that low-priority device can be starved. In order to prevent starving, cyclic priority assignment will be used when appropriate.
Unlike static assignment of ARB line per device, in cyclic priority ARB line assignment is changed every clock in cyclic manner. Thus, at any given time the arbitration priority of all devices is random, which effectively provides equal priority for each device. In order to avoid collisions, each arbiter is initialized with distinct priority and all arbiters are fully synchronized.

Data transfer phases

Upon receiving of NGIO cell (or upon translating PCI cycle to NGIO cell), the data transfer starts. Each data transfer goes through following steps (phases):

Phase1 – MAC translation

Receive queue extracts MAC address from accepted cell and translates it to port address using FDB table. It is possible to have a local copy (or cache) of FDB table or inquire FDB unit for translation. ***FBUS*** implements this phase.

Phase2 – post transmit request

Receive queue arbitrates request bus (***TREQ***) and posts transmit request to the transmit queue.

Phase3 – transfer data to transmit queue

Transmit queue requests data transfer from receive queue, and data is transferred on data part of ***DiBUS***. ***TRQ*** and ***DiBUS*** busses implement this phase.
Upon completion of one phase, the transaction may wait in the queue for unlimited time till its next phase is scheduled.
In heavy load environment each transaction is expected to go through all three distinct phases. In light load performance optimizations are made to boost transfer. Under certain conditions that are discussed later, phases can be merged. Refer to Data transfer summary section for details

Phase1 – MAC translation

FBUS protocol implements the phase1 of data transfer. Figure 5 illustrates ***FBUS*** cycle implementing first phase of data transfer protocol.

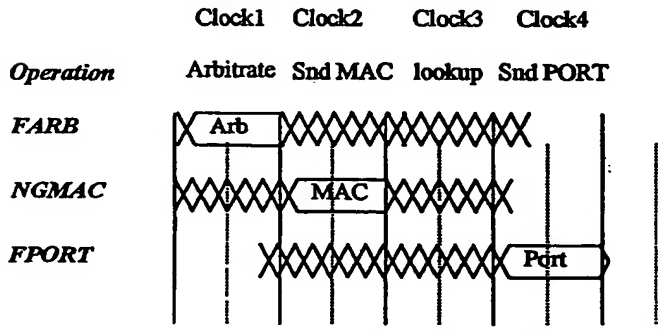


Figure 5 - MAC to PORT translation cycle

In clock1 *NGMAC* was arbitrated and ownership was granted for the next cycle. In clock2 MAC address is sent to FDB on the *NGMAC* bus. It is assumed that full clock is needed to send the MAC to FDB. Table

lookup is performed in the next cycle (clock3), and in clock4 port address and its speed is returned on *FPORT* lines.
Target queue port ID will be used by the receive queue to post data transfer request to transmit queue, it will be inserted to *TQID* field of the ***TREQ*** bus in transmit request phase.
The receive queue must examine the priority of the cell. If cell priority equals 15, then it should be routed to FSA port, regardless of port number replied by FDB. This is done in order to assure that FMP will not be discarded in case of the receive queue overflow.

Phase2 – Post transmit request

TREQ protocol implements second and third phase of data transfer – post transmit request and transfer data to transmit queue.

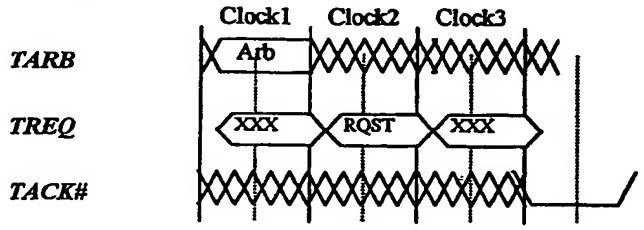


Figure 6 - acknowledged transmit request cycle

Figure 6 illustrates acknowledged transmit request phase. In clock1 ***TREQ*** bus is arbitrated with ***TARB*** bus, and transmit request is placed on ***TREQ*** bus in clock2. In clock4 *TACK#* signal is asserted by the transmit queue, indicating that request has been registered.

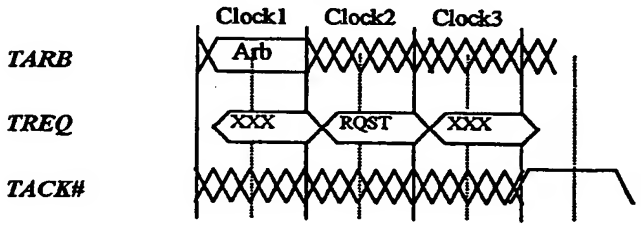


Figure 7 - denied transmit request cycle

Figure 7 illustrates denied transmit request phase. In clock1 ***TREQ*** bus is arbitrated with ***TARB*** bus, and transmit request is placed on ***TREQ*** bus in clock2. In clock4 *TACK#* signal negated by the transmit queue, indicating that request has not been registered, and receive queue needs to start transmit request phase over again.

Phase3 – Data transfer

Data transfer protocol concludes the transaction. Target queue requests data transfer from the receive queue using ***DiREQ*** bus. Receive queue transmits data to the target on ***DiBUS***.

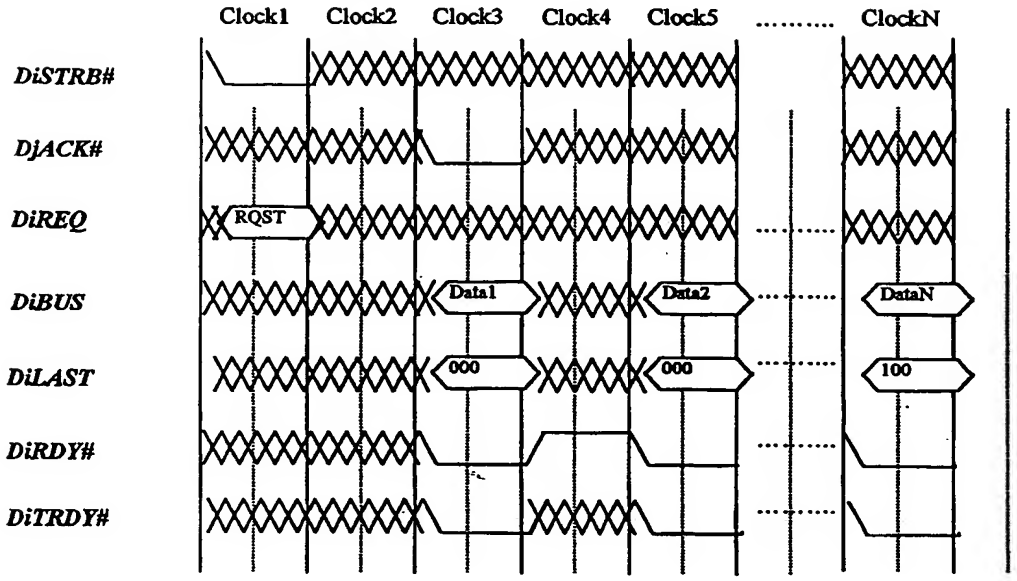


Figure 8 - successful data transfer phase

Figure 8 illustrates successful data transfer phase. In clock1 transmit queue places request for data transfer from the receive queue specified in *receive queue ID* field of ***DiREQ***, qualified by ***DiSTRB#***. All receive queues snoop ***DiREQ*** bus and one whose port number matches *RQID* field of ***DiREQ*** replies with ***DjACK#*** signal, assumes ownership of ***DiBUS*** and ***DiRDY#*** busses two clock after data request (clock3) and sends data to the transmit queue. The ownership of ***DiBUS*** and ***DiRDY#*** starts from the next cycle (clock3), and target queue should be ready to accept the data starting from clock3. Receive queue can reject the request from transmit queue by negating ***DjACK#***. This will be done in case all output ports of the receive queue are busy (e.g. it is already transmitting to all directions). Once receive queue acknowledged the transmit queue request, it must send the first chunk of data no later than 5 clocks after transmit queue request was acknowledged.

Receive queue qualifies data transfer with ***DiRDY#*** signal. ***DiLAST*** signal is driven with '000 value in clock 3 and 5, indicating that it is not a last data transfer of the cell. ClockN is that last clock of cell transfer with all four ***DiBUS*** bytes valid, as indicated by the ***DiLAST*** value '100. Note ***DiTRDY#*** is asserted after every data transfer on ***DiBUS***, indicating that transmit queue accepted the data sent by receive queue.

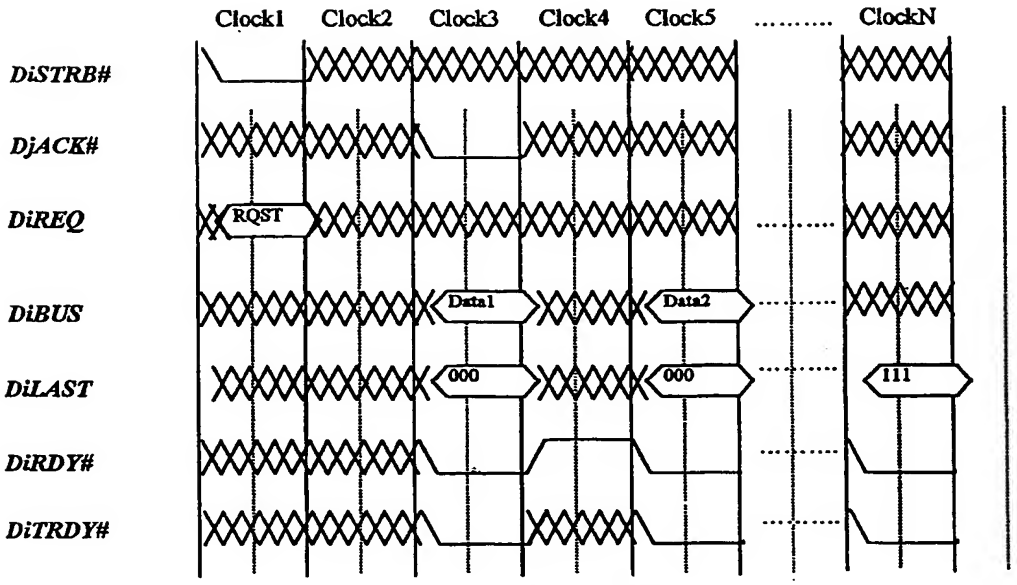


Figure 9 – data transfer completed with error

Figure 9 illustrates data transfer phase that terminates with error. In ClockN '111 value is placed on *DiLAST* signals by source queue, indicating that transmit of this cell should be terminated with Error Propagation Character (EP). Note that in this case value of *DiBUS* is ignored, and transmit queue appends EP character to the cell being transmitted. In case of error cell termination, the actual length transmitted by the receive queue can be below minimal cell length. Transmit queue must pad the short cell with dummy data while transmitting to assure that no illegal cell is generated by the MT101.

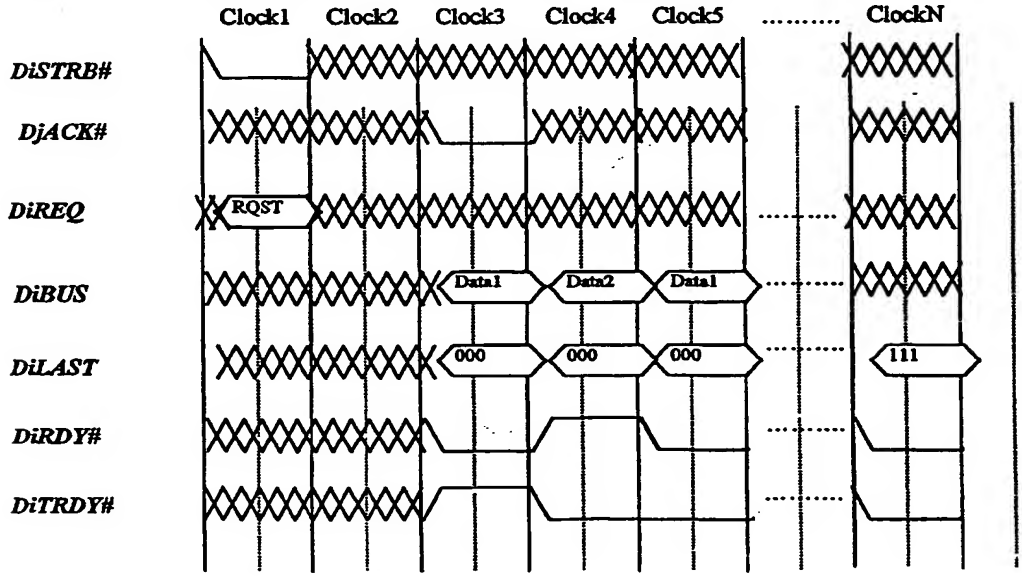


Figure 10 – re-try in data transfer

Figure 10 illustrates data transfer re-try. In clock3 valid data was placed by the receive queue on the bus (*DiRDY#* asserted), but it was not accepted by the transmit queue, as indicated by *DiTRDY#* being inactive in this cycle. Receive queue re-sends all data starting from the chunk that was negated by the target queue.
Figure 11 illustrates rejected data transfer request.

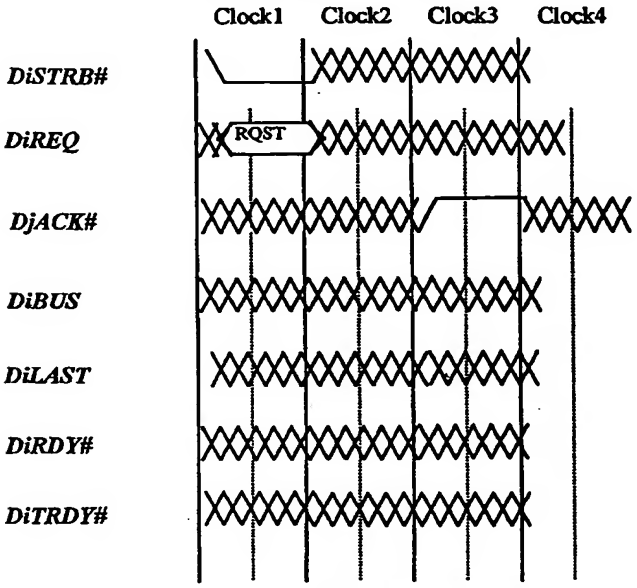


Figure 11 – rejected data transfer request

Data transfer request posted on *DiREQ* bus in clock1. In clock3 *DjACK#* was negated by the receive queue, which indicates to transmit queue that data transfer cannot start within a committed time limit, and therefore transmit queue needs to request data transfer again later. Note, that once data transfer request was negated by the receive queue, it means it did not assume responsibility to drive *DBUS* lines (*DiBUS*, *DiRDY#*) – e.g. receive queue that owned these lines must keep driving them in clock4 to avoid leaving them floating.

Data transfer summary

This section will show full cycle of data transfer – from the clock MAC is available on the receive queue till first cycle of data transfer.

Three-phase data transfer

This section shows full cycle of data transfer that goes explicitly through all three phases.
The first event of the data transfer is arbitration for *NGMAC* bus. This can be done in the same cycle MAC being extracted form the header.
Figure 12 illustrates an explicit-phase inter-unit data transfer.

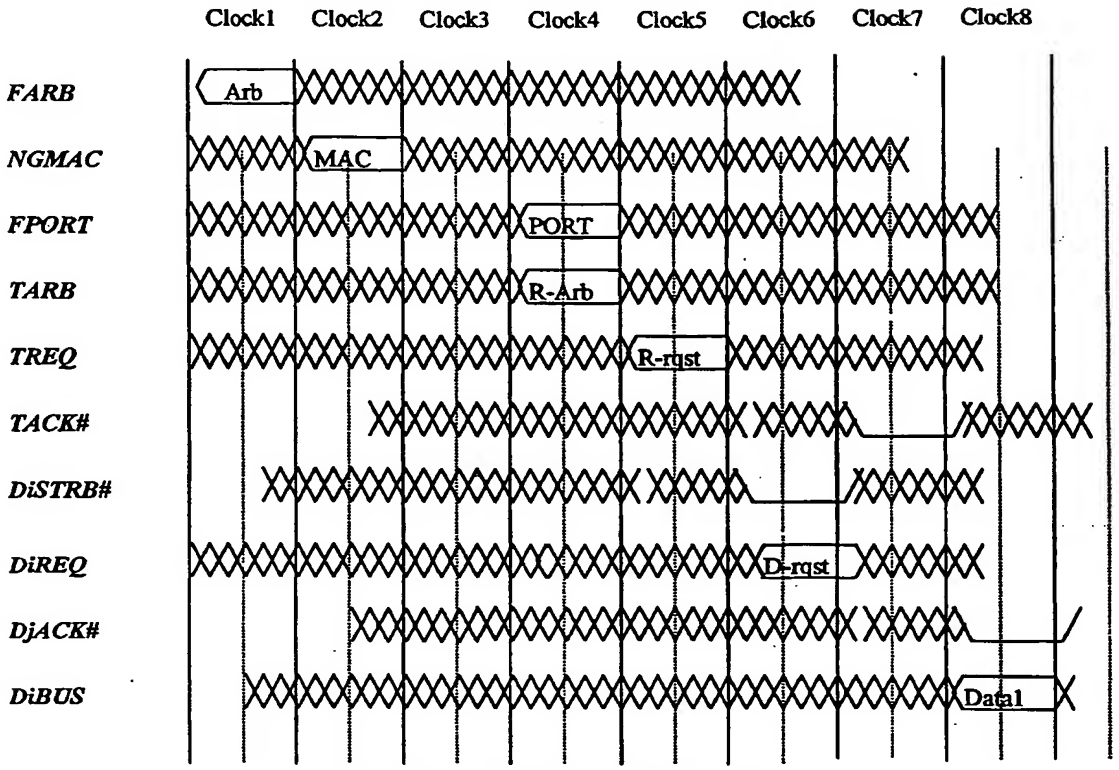


Figure 12 – 3-stage inter-unit data transfer

Inter-unit data transfer period starts in clock1, where receive queue arbitrates the *NGMAC*, which is granted for the next cycle. In clock2 MAC address is sent to FDB, in clock3 FDB lookup is performed and in clock4 PORT value received from FDB, which concludes the first phase of data transfer. This phase can be shortened if receive port contains FDB lookup table (or some sort of FDB caching), which saves flight time on the busses.

Second phase of data transfer starts after first phase was concluded. With limited pipelining, second stage can start not earlier than in the clock PORT was returned to the requesting unit. The first event of the second stage is *TREQ* arbitration by receive queue (clock4). *TREQ* is granted for the next cycle, and data transfer request is placed on *TREQ* bus in clock5. In clock6 transmit request acknowledged (*TACK#* asserted), which concludes second phase of data transfer.

Third stage of data transfer starts after second phase was completed. With limited pipelining, third stage can start not earlier than the clock transmit queue acknowledges the transmit request. The third stage starts when transmit and places data transfer request on *DiREQ* lines in clock6. In clock8 the first chunk of data can be driven by receive queue, qualified by *DiRDY#* and *DiTRDY#* (not shown on the figure).

Two-phase data transfer

Under certain conditions different phases of data transfer can be merged. If target queue is idle, the data transfer can start already in phase2, hereby merging phase2 and phase3 of data transfer.

Each transmit queue indicates that it is empty by asserting *TQiIDLE#* signal. Each receive queue observes all *TQiIDLE#* signals, and if data is targeted to idle transmit queue, data transfer can start in the same clock transmit request is placed on *TREQ* lines (clock5), reducing latency by 3 cycles.

Figure 13 illustrates the case where phase2 and phase3 of data transfer are merged.

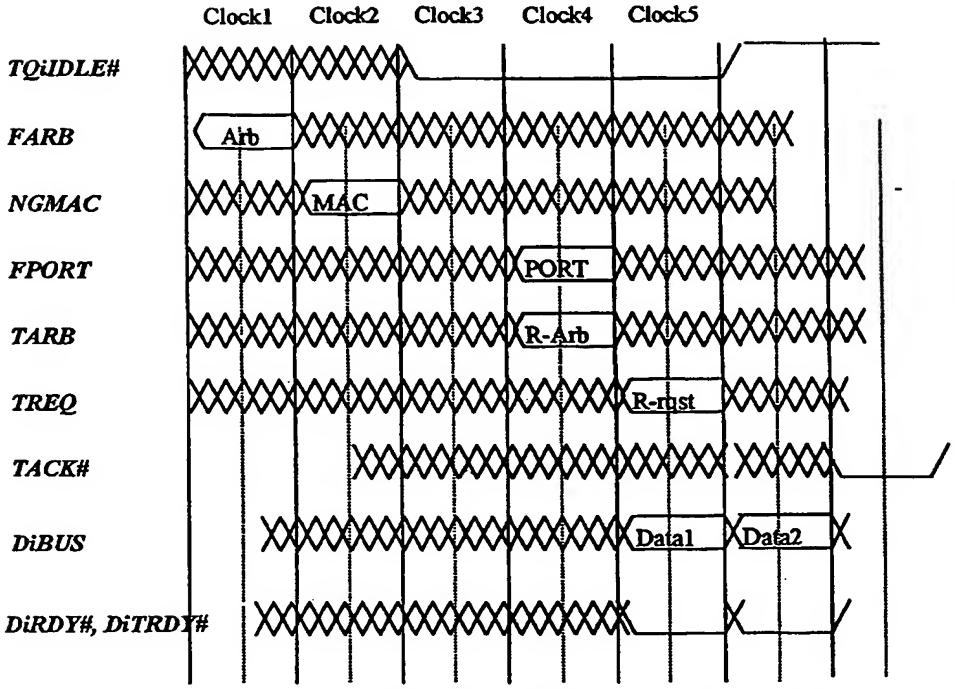


Figure 13 – two-phase data transfer (phase2 and phase 3 merged)

In clocks 1-3 MAC translation is performed as described above. While arbitrating for ***TREQ***, receive queue observes that merge conditions for phase2 and 3 met (refer to section), and thus it places first chunk of data on respective ***DiBUS***, qualified with ***DiRDY#*** in the same clock is places the transmit request on ***TREQ*** bus. Target queue asserts ***DiTRDY#***, acknowledging the data receive. Only if phase2 and 3 conditions are met, receive queue is allowed to drive data on the bus in the clock ***TREQ*** is placed. If merge condition occurred, transmit queue will not place request on ***DiREQ*** for this data transfer anymore.
Every time receive port places request on the ***TREQ***, it must observe target ***TQiIDLE***, as transmit request can be placed on ***TREQ*** bus in the same clock respective transmit queue asserts ***TQiIDLE***, which triggers the phase merge conditions.
NGIO to NGIO latency can be improved further by implementing FDB lookup table in each receive queue or by caching most frequently used entries. See PCI to NGIO transfer discussion for details. This option will be evaluated at the later stage.
Note that receive queue speed is slower than transmit queue, receive queue needs to pay extra caution while merging phases. It can be done in two ways:

1. After receiving destination port information, nullify (force NOP) ***TREQ***. Don't attempt to arbitrate until enough data is buffered in the queue. This is preferred option from performance standpoint.
2. Delay arbitration for ***TREQ*** by one cycle and start arbitration only after enough data is buffered.

The choice between the two options is implementation-dependent.
If FDB is cached on the receive ports, phase1 of data transfer protocol can be performed internally on the queue, now showing up on the ***FBUS***. In this case data transfer latency will shorten by two cycles. MAC address of PCI will always be cached on every port, so transfers to PCI will take three cycles.

Performance summary

Table 1 below summarizes best case latency for data transfers.

Transfer	clocks
NGIO to NGIO	5
NGIO to PCI	3
PCI to NGIO	3

Table 1 - minimum data transfer latency

Fabric Management data transfers

The Fabric Service Agent (FSA) appears as NGIO port to the internal protocol with port address 9 (nine) – the largest port number. All accesses to port9 will access FSA.
FSA will contain the Fabric Management Packets Queue (FMPQ), and data will be transferred to the queue on *D9BUS* using data transfer protocol. If priority in the cell received by NGIO port is 15, it is FMP cell, and should be routed to the FMPQ.
Transfers to FMPQ are similar to transfer to any other port. Data is queued in the receive queue, request is posted to FMPQ (phase2). Eventually FMPQ will request cell transfer, and data will be transferred to FMPQ.
If FMPQ is empty, FSA will assert corresponding *TQiIDLE*, and newly arrived cell can be transferred from the NGIO port to FSA even if its data array is full. Each receive queue should preserve a bus driver it its data array is full and there is no pending FMP in the array. If there is no room in receive queue to place FMP arrived and FSA has FMP in process ('phase2 and phase3 merge' condition is not met), the FMP should be dropped by the receive queue.

Configuration registers access

Configuration registers are accessed using *CBUS* bus bundle, which contains of *CRBUS, CADS#, CRW#, CRSRC* and *CRDY#* signals. All configuration reads/writes are initiated by FSA, S-EPROM, PCI or 8-bit CPU interface unit and each unit responds to its registers' access per address specified in table yyy below. Configuration register (CR) address is driven on *CRBUS* in the address phase of the cycle, qualified with *CADS#*. Read/Write# and request source indication is driven with address on *CRW#* and *CRSRC* lines respectively. Each configuration register can be accessed from PCI, NGIO and CPU. While being accessed from its 'natural' source (e.g. PCI configuration from PCI port), unit holding the register must obey access rules as specified (e.g. protect read-only registers from being written etc). While being accessed from 'other' source (e.g. PCI configuration registers from CPU), all register should behave as regular data storage – e.g. all bits are written on write operation and read on read operation.
CRBUS is not pipelined, there can be no more than one active cycle at a time, which assures no contention on the bus.
All registers are 32-bit registers, e.g. each *CRBUS* operation consists of one-clock address phase and onetwo-clock data phases. Figure 14 illustrates CR read and CR write operation.

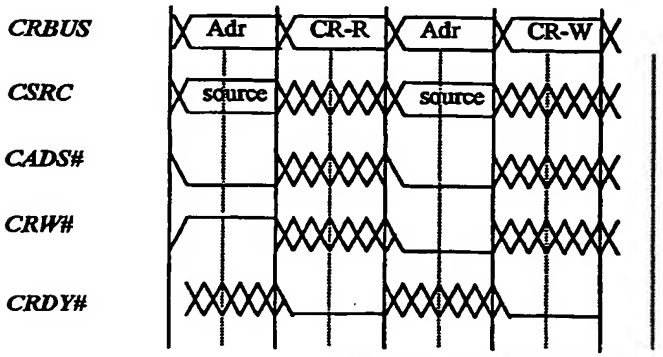


Figure 14 – CR read and write operation.

Configuration registers access can be initiated by CPU, S-EPROM, from PCI or from FMP.
FMP-originated accesses are performed by FSA, which breaks FMP into series of *CBUS* cycles per FMP

received. FSA contains CBUS arbiter, implementing '*HOLD/HLDA'* arbitration protocol. *HOLDi* used to force unit out of the *CBUS, HLDAi* acknowledges bus release. Figure 15 illustrates *CBUS* arbitration.

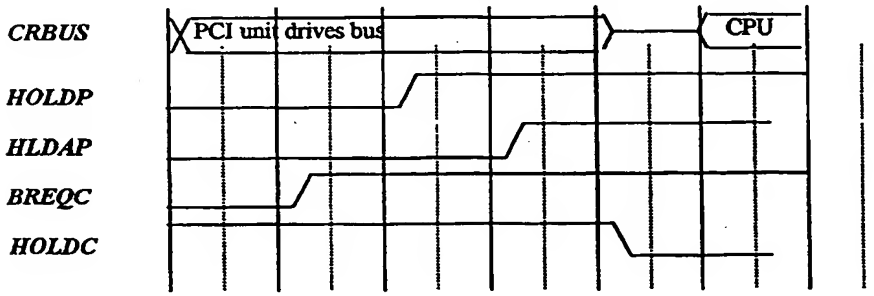


Figure 15 – CBUS arbitration

The figure shows example of CBUS arbitration between PCI and CPU units. At the beginning PCI owns *CBUS* and drives it. CPU units asserts *BREQC* signal, indicating that it needs *CBUS* to access the registers. Next clock arbiter asserts *HOLDP*, forcing PCI unit off the bus, and PCI unit acknowledges with *HLDAP*. Clock after it asserts *HLDAP*, it quits driving the bus. Arbiter clears *HOLDC*, and from now on *CBUS* is driven by the CPU unit

Cells squash (discard)

Due to various conditions, cells can be discarded by the MT101. Cells can be discard condition can be triggered in receive queue (invalid port etc) and in transmit queue (lifetime expired). Cells are discarded due to following conditions:

1. Invalid (garbage) target port. This condition is received from FDB at MAC translation phase.
2. Cell lifetime expiration – cell lifetime exceeded.
3. Error encountered in cell CRC – this will be done (if at all) in the receive queue.
4. Another conditions that I cannot think about now

The cell can be discarded by receive queue any time before transmit queue requested data from receive queue (e.g. any time before phase3 of data transfer started). If cell discard condition occurred before transmit request posted, the cell is discarded without any external notification. If cell discard condition occurred after transmit request posted to transmit queue, receive queue needs to notify transmit queue to cancel the transmit request. This is done by posting transmit request for the same cell for a second time to the same transmit queue with 'discard' opcode on *TREQ.CMD* lines. Transmit queue acknowledges removal of transmit request with *TACK#* signal. If cell squash request posted after data transfer started, transmit queue will deny this request (by negating *TACK#*), and receive queue will send the data to transmit queue.

If cell discard condition occurred after third phase of data transfer started (data transfer request acknowledged by the receive queue), receive queue can either cut the transmission by terminating data transfer with *ERROR* delimiter (forcing *DiLAST* signals to '111) or complete the transmission.

If cell discard condition occurred in transmit queue (life timeout), transmit queue notifies respective receive queue on *DREQ.CMD* field that specific cell needs to be discarded. This request implies implicit acknowledge from the receive queue (e.g. receive queue cannot reject or postpone such a request).

Transmit queue has a timer (counter) for each priority queue. When new entry reaches the top of the queue, the counter is loaded with LifeTime value of PortInfo COD field. On transition from '1 to '0, the discard event occurs and transmit queue posts discard command to the respective receive queue, removes killed entry from the head of the queue, places new entry to the queue and re-initiates the lifetime counter.

Data transfer responsibility

Once MT101/102 assume responsibility of the data transfer from PCI bus, it is guaranteed that data transfer will be completed within (programmable) period; else error will be signaled to the SW. In order to assure

data transfer completion, NGIO channels with PCI end-points must be connected/acknowledged channels. The data transfer ownership of MT101/102 is implemented through rules defined below:

1. All channels with PCI at the end point must be connected/acknowledged channels.
2. On memory reads PCI unit transfer the cycle to delayed read. PCI unit logs the read request (address, CMD, byte enables etc.) and generated RDMA-read NGIO cell. Subsequent retries for the same cycle by the host will be delayed by PCI unit until RDMA-read response with data arrival, when data will be buffered and sent to the host after subsequent retry.
3. I/O reads treated same way as memory reads.
4. Non-posted writes and I/O writes treated same way as memory reads. In addition to address and CMD, PCI unit also logs data written, and generates RDMA write NGIO cell, specifying that it is posted (I/O) write. After acknowledge received, PCI unit will compare data of re-try cycle, and if address, CMD and data matches original cycle, PCI unit returns TRDY# and completes cycle.
5. Posted writes generate RDMA-write NGIO cell, and returns TRDY# immediately to the host. The NGIO cell ID (MAC, PSN etc.) is kept alive in PCI unit until RDMA-write is acknowledged.

If acknowledge for the cell does not arrive within specified period (refer to configuration section), the event is logged, and can further generate interrupt to the Master Fabric Manager.

[1] The most challenging case here is to assure that rule 3 for 'transaction ordering and posting for bridges' is enforced (page 42 of PCI spec)

[2] The priority of the cell is not taken here into consideration for simplicity. NGIO fabric will take care about it and high-priority cells will pass over low-priority ones.

Global signals and events definition

Protocol events

Event	Condition
Phase2 and Phase3 merge	In the clock of arbitration for *TREQ*: 1. *TQiIDLE#* is asserted AND 2. no valid *TREQ* for the target port is snooped on *TREQ* in that clock
TQiIDLE# assert	Clock before transmit port can unconditionally receive new cell data (the earliest)
TQiIDLE# negate	Clock after first data chunk was driven to the port.

Table 2 - Global events summary

Global signals' summary

Table below summarizes global signals and busses of the device. Timing of each bus is specified as

1. Early (driven from the latch, available early in the cycle),
2. Medium (goes through several logic gates, available in the middle of the cycle)
3. Late (goes through significant logic before driven out, available in the late part of the cycle, should be sampled without excess logic load).

Bus name	Description	#	wires
FARB[8:0]	Arbitration bus, used by all ports to arbitrate for *NGMAC* bus and FDB port for MAC to port translation. The bus arbitration is done as described in Arbitration Protocol section. Bits 8:0 are assigned to the remaining ports and deploy cyclic priority arbitration.	1	9
NGMAC[17:0]	This bus is used to send MAC address for translation, and it contains 2 filelds: *MAC* – NGMAC[15:0] – MAC address *Rspeed* – NGMAC[17:16] – Receive queue speed	1	18
FPORT[6:0]	Used to send the target port number and its speed from FDB to receive queue. If *FPORT6* bit of *FPORT* is clear, *FPORT[5:0]* contains valid port address and speed. If *FPORT6* is set, *FPORT[5:0]* encodes special cases: '100000 – discard cell '1xxxxx – reserved For more details please refer to FDB unit spec. *Port* – *FPORT[3:0]* – port address *Buffer* – *FPORT[5:4]* – Amount of buffering required before TREQ (Error! Reference source not found.)	1	7
TARB[9:0]	Arbitration bus, used by receive queues to arbitrate *TREQ* bus. The bus arbitration is done as described in Arbitration Protocol section. *TARB9* (the most significant bit, corresponding to highest priority) is always assigned to the PCI port, which assures highest priority for PCI. Bits 8:0 are assigned to the remaining ports and deploy cyclic priority arbitration.	1	10

Bus name	Description	#	wires
TREQ[36:0]	Request bus, used in phase2 of data transfer to post transmit request. *TREQ* bus has following fields: *CMD – TREQ[2:0]* decodes command to be executed by ports. Following commands are supported: 000 – NOP. All fields of TREQ bus should be ignored. No bus drive responsibility changed. 001 – Discard cell. Used by receive queue to discard transmit request that was posted already to transmit queue. 010 – transmit request. Driven by receive queue while posting request for transmit queue in second phase of data transfer protocol 011, 1xx – Reserved. *TPID – TREQ[6:3]* contains port number of the queue this request is targeted for. All transmit queues must snoop *TREQ* bus one clock after it was arbitrated and if *TPID* matches port number mapped to the queue, it must respond to the request (acknowledge/deny). *RQID – TREQ[10:7]* contains Receive Queue ID (port number). *Priority – TREQ[14:11]* contains cell priority after re-map from NGIO to MT101 priority queues.. Valid values are 0,1,2,3 and 15 *CellID – TREQ[19:15]* contains ID of the cell in the receive queue. *DataAdr – TREQ[28:20]* contains address of this cell in receive queue data array *Opcode – TREQ[36:29]* contains opcode field from the NGIO cell.	1	37
TACK#	Indicates that request posted on *TREQ* in one before previous cycle was accepted.	1	1
DiSTRB#	*DiREQ* bus strobe. If *DiSTRB#* asserted, valid data request is posted on *DiREQ* bus.	1	10
DiREQ[19:0]	Request bus, used in and phase3 of data transfer. Each transmit queue has *DiREQ* bus associated with it. Each receive queue observes all *DiREQ* busses *DiREQ* bus has following fields: *CMD* – DiREQ[1:0] contains to command being sent to the receive queue: '00 – data transfer '01 – discard cell '1x - reserved *RQID – DiREQ[5:2]* contains Receive Queue ID (number). All receive queues must snoop *DiREQ* bus. If *RQID* field matches the receive port number, then it is a request for data transfer from receive queue to transmit queue. *CellID – DiREQ[10:6]* contains ID of the cell in the receive queue. *DataAdr – DiREQ[19:11]* contains address of the cell in receive queue data array	10	200
DjACK#	Indicates that request posted on *DiREQ* in one before previous cycle was accepted. Each receive queue has its own *DjACK#* signal. Transmit queue that requests data from the receive queue should observe respective *DjACK#* clock after data request was posted on *DiREQ* bus.	1	10
TQiIDLE#	Indicates that target queue *i* is idle and can receive cycle unconditionally. Exact timing of this signal is specified in Key Events section.	10	10

Bus name	Description	#	wires
DiBUS[15:0]	Data bus, used to transfer data from receive queue of any device port to transmit queue of port *i* (the port associated with this bus).	10	160
DiLAST[2:0]	Indicates status of the current data transfer. Using the following encoding: 000 – both bytes of *DiBUS* are valid and more data corresponding to this cell is expected to be transferred by receive queue that currently drives the *DiBUS* 001 – last transfer of the cell with one valid byte. 010 – last transfer of the cell with two valid bytes. 101 – last transfer of the cell with error. The cell transmit should be terminated with Error Propagation Character (EP), one byte is valid on *DiBUS* 110 – last transfer of the cell with error. The cell transmit should be terminated with EP. Two bytes are valid on *DiBUS* 111 – cell terminated with error, EP should be appended to the cell on transmission. Both bytes on *DiBUS* are invalid (e.g. no data transfer, only error indication) 100, 011 – reserved	10	30
DiRDY#	Signal indicating that corresponding *DiBUS* and *DiLAST* have a valid data driven by receive queue	10	10
DiTRDY#	Signal indicating that transmit queue has accepted (latched) *DiBUS* and *DiLAST* values from the respective busses. If *DiTRDY#* is inactive, receive queue must re-transmit the data starting from the chunk that was not accepted by the target queue within a given number of clock cycles. To avoid deadlocks (*DiRDY#* and *DiTRDY#* oscillating), once *DiTRDY#* is asserted, it cannot be cleared before valid data chunk arrived (similar to PCI bus *TRDY#* rules). *DiTRDY* qualifies 4-chunk transfer. Transmit queue can negate data transfer only on the 4-chunk boundary (e.g. negate only first chunk). DiTRDY will be ignored by the receive queue during transfer of the 2nd, 3rd and 4th data chunk.	10	10
DiBSY	Signal indicating that data is driven on *DiBUS* by receive queue. *DiBSY* signal is cleared clock after *DiLAST*, and asserted if *DiBUS* is committed to receive queue. For back-to-back transfers (when TxQ requested new data before current transfer is completed on *DiBUS*), *DiBSY* will be cleared for a single clock. In the clock *DiBSY* is cleared, TxQ owns (drives) *DiBUS*	10	10
RQjSTAT[3:0]	Request queue status, provides auxiliary information that can be used by transmit queue for arbitration decision. Usage of this bus is not mandatory, it can help to improve throughput and system utilization in high loads. *RQjSTAT* bus has following fields: *Fchan – RQjSTAT[1:0]*. This field indicates number of channels free for data transfer in receive queue. If *RQjSTAT.Fchan* = '00, it means that all channels are busy with data transfer and request for data transfer will be denied (refer to Phase3 – Data transfer section) *FC – RQjSTAT[3:2]*. This field indicates flow control issued by the port (folded to MT101 priorities)	10	40
CRBUS[31:0]	Bus used to read/write control and configuration registers of the device. Refer to Initialization and configuration section for protocol	1	32
CADS#	Indicates that valid configuration register address is placed on *CRBUS*. Initiates CR access	1	1
CRW#	Indicates whether CR access is read or write.	1	1

Bus name	Description	#	wires
CRDY#	Indicates that valid CR data is placed on *CRBUS* for CR reads. Indicates that data placed n *CRBUS* for writes has been sampled by the target.	1	1
CRSRC[2:0]	Indicates source of register access: 000 – access initiated from PCI port 001 – access initiated from NGIO port 010 – access initiated from 8-bit CPU port 011 – access initiated from Serial EPROM 1xx – Reserved.	1	3
HOLDC, HLDAC, BREQC	*HOLD/HLDA/BREQ* signals for CPU unit, used for *CBUS* arbitration.	1	3
HOLDE, HLDAE, BREQE	*HOLD/HLDA/BREQ* signals for EPROM unit, used for *CBUS* arbitration.	1	3
HOLDP, HLDAP, BREQP	*HOLD/HLDA/BREQ* signals for PCI unit, used for *CBUS* arbitration.	1	3
INIT_PCI	Initialization process being performed, PCI unit should back-off (retry) all cycles	1	1
INIT_NGIO	Initialization process being performed, NGIO should not monitor/establish link	1	1
TCLK	Timeout Clock – all NGIO timeouts should be counted in this clock	1	1
RESET	Global HW reset	1	1
			6390

Table 3 - global signals summary

NGIO port external definition and requirements

NGIO port consists of the following basic blocks:

1. Receive queue – responsible to receive NGIO cell, inquire FDB to translate MAC to PORT address, notify appropriate transmit port and deliver data upon transmit port request. Receive queue simultaneously transmit upto 4 cells to 4 different transmit queues.
2. Transmit queue – responsible to accept transmit requests from receive queues, arbitrate the link and request data to be transferred from the receive queue.
3. Link maintenance machines – responsible to maintain NGIO link, generate delimiters, identify link failure.

Link maintenance machines – details.

Link check machines are responsible to maintain link, identify link existence (good link), synthesize and transmit adequate control characters.
After initialization sequence is completed (*INIT* signal cleared), the Link Machine establishes channel connection at speed specified in the port configuration register.
Receive Link machine identifies delimiters and notifies the Receive queue when new cell arrives. Transmit link machine generates delimiters between the cells.
Link machine checks the link status as specified in NGIO Link document. In case of link failure, link machine sets LinkDown bit in the port status register. This register can be read by SW (through FMPs or from CPU side) and by HW (FSA, transmit queue).

In case of link disconnect, all flow control from this link should be removed. Transmit queue will squash all data sent to it, hereby acting as /dev/null – this is in order to flush all cells pending transmission to that port.

Receive queue - details

NGIO receive queue is responsible for

1. accept NGIO cells
2. initiate FDB inquire to translate MAC address to port number
3. request data transfer from target port. Requests should be sent to target port in order of their arrival within same priority to avoid illegal out-of-order transmission. Higher priority cells should be scheduled for transfer before low-priority ones.
4. deliver cell data to the target port upon request
5. keep track of cell's age and squash expired cells
6. issue flow control messages to avoid queue overflow by inbound traffic
7. Check cells for errors (CRC) and inject EP as necessary (not a MUST per switch spec, but good feature to debug the network).
8. Squash cells that arrived with EP delimiter (configuration).

Receive queue should fully implement data transfer protocol (all phases).
Receive queue consists of two major blocks

1. Data array. Data array stores cells that were received by the transmit queue. Array size is 292x7 bytes, e.g. it can contain upto 7 NGIO cells of maximum length.
2. Cell pointers. This is a register file of 16 entries, which contains pointers to cells in the array. The pointers contain all cell information needed to post request, make decision about cells' squash etc.

Receive queue should generate flow control, which can have two distinct sources:

1. The data array runs out of space. Flow control will be issued based on space left, the watermarks are programmable, A1…A8 specifies number of left empty in the array. A1<=A2<=…<=A8. The values of A1…A8 are programmable through the receive queue control register.

Number of empty entries	Flow control
A1	XN8
A2	XN7
A3	XN6

Number of empty entries	Flow control
A4	XN5
A5	XN4
A6	XN3
A7	XN2
A8	XN1

Table 4 - Flow control conditions - data array

2. Receive queue runs out of cell pointers. In this case flow control should be issued according to the rules summarized in the Table 5. N1<=N2<=...<=N8. N1...N8 values are programmable in the receive port control register

Number of empty entries	Flow control
N1	XN8
N2	XN7
N3	XN6
N4	XN5
N5	XN4
N6	XN3
N7	XN2
N8	XN1

Table 5 Flow control conditions - pointers

Transmit queue - details

NGIO transmit queue is responsible for:
1. accept data transfer request from receive queue of (other) NGIO port
2. acknowledge the acceptance of data transfer request
3. resolve priority of all pending transmit requests
4. inquire data for transmission from the corresponding receive queue

Transmit queue should log requests from the receive queues and arbitrate the outbound link. Transmit queue should collect enough information to make a right decision during second phase of data transfer (logging requests from the receive queues). Transmit queue will take into consideration following constrains while arbitrating the link (in priority order):
1. Priority of outstanding requests. Higher priority request should be transmitted first
2. Load on the receive queue. Requests from receive queue with higher load should be served before requests from queues with lower load
3. Non-starving of low-priority requests, implementing LiveLock register.

More information can be collected by the transmit queue (like cell length etc.). It is not clear at this point whether additional information is necessary.
In case of link disconnect, all flow control from this link should be removed. Transmit queue will squash all data sent to it, hereby acting as /dev/null – this is in order to flush all cells pending transmission to that port.

PCI port external definition and requirements

MT101 PCI port supports up to 66MhZ PCI bus. MT101 supports 32 and 64-bit PCI with full 64-bit address space support.
Once PCI slave assumes responsibility on the cycle, it is responsible to assure its completion, else error (interrupt) will be issued. Hence, all PCI-originated channels are connected/acknowledged channels – no exceptions.
PCI port consists of the following basic blocks:

1. NGIO port associated with PCI port, implementing interface of PCI to NGIO world.
2. PCI bus master block – responsible to translate requests from NGIO network to PCI and return the response back to NGIO world
3. PCI slave block – responsible to accept PCI requests, translate them to NGIO world by issuing appropriate NGIO cell, accept response from NGIO network and translate it back to PCI world.

NGIO port – details

NGIO port is designed in such a way that it can interface to PCI block, so it can be used in PCI unit almost without changes. PCI unit interface to NGIO port similarly to the way NGIO link interface.
Since both master and slave of the PCI may have same MAC address and hence same port address, the transmit queue of NGIO port associated with PCI needs to take ***TREQ.Opcode*** field into consideration while responding to the ***TREQ*** request. If ***TREQ.Opcode*** contains response opcode, the request is targeted to the PCI slave. Otherwise it the cell is targeted to the PCI master unit.
In order to assure PCI cycles ordering is preserved[1], following rules must be followed[2]:

1. Cycles should be sent to NGIO fabric in same order they are issued on PCI bus.
2. Cycles received from NGIO fabric should appear on PCI bus in same order they received form the fabric.

PCI master and slave units will assure that requests are ordered before entering the common receive queue, see PCI master and PCI slave MAS for details.
The ordering of received cells will be assured by the common transmit queue, which will not accept new cell from NGIO fabric unless it is 'safe' to forward it to the PCI master or slave. Table 6 defines condition for transmit queue whether to accept incoming cell – depending on its opcode and status of the PCI master.
PCI master status indications:
WP – Write Pending. Write request accepted from NGIO fabric, but write cycle on the PCI bus has not been completed. Master can accept additional writes
WF – Write buffer full. PCI master write buffers are full, master cannot accept additional write requests
RF – read buffer full. PCI master cannot accept additional read requests.
Possible NGIO cells arrive to the PCI transmit queue:
WR – RDMA-write request (opcode '110 through ''1011)
RD – RDMA-read request (opcode '1100)
RR – RDMA-read response (opcode '1101 through '10000)
ACK – Acknowledge response (opcode '10001)
Table 6 summarizes conditions when transmit queue will ask for a respective NGIO cell to be transferred to the PCI unit.

Cell\Status	WP	WF	RF	Otherwise
WR	Yes	No	Yes	Yes
RD	No	No	No	Yes
RR	No	No	Yes	Yes
ACK	Yes	Yes	Yes	Yes

Table 6 - PCI transmit queue data request conditions

[1] The most challenging case here is to assure that rule 3 for 'transaction ordering and posting for bridges' is enforced (page 42 of PCI spec)
[2] The priority of the cell is not taken here into consideration for simplicity. NGIO fabric will take care about it and high-priority cells will pass over l w-priority ones.

The rules above assure that cycles will appear on PCI bus in the same order they were received form the NGIO fabric. However, due to out-of-order nature of PCI, they can be completed not in order they were issued (e.g. PCI read cycle was delayed, and subsequent write, was completed). In this case – although cycles were completed out of order on the PCI bus – the PCI master should send acknowledges back to the fabric in right order. In other words, if some PCI cycle completed before its predecessor, the acknowledge to that cycle should be held off by the PCI master till the first cycle is completed.
This mechanism will assure that PCI ordering rules as defined in Appendix E of PCI specification are obeyed.
Flow control is generated by the Receive Queue using same algorithm as 'normal' receive queues. The Flow Control is being considered by PCI slave – it stops accepting (e.g. re-try without log) of priorities lower than Flow Control issued but the receive queue. Transmit queue stops accepting (DREQ) of requests to master with priority lower than Flow Control priority. Cells targeted to the slave with priority lower than flow control can be accepted by transmit queue, as long as ordering is kept within a channel.
Master ignores flow control issued by the receive queue. All outstanding requests can be returned by the master despite FC – this is in order to avoid ordering problems.

PCI master – details

PCI master is responsible to translate NGIO cells targeted to PCI to PCI cycles, issue the cycles to PCI, generate response cell and send it back to NGIO network.
Following requests (NGIO cells) should be served by the PCI master:

1. RDMA-read. Master should issue read on the PCI bus with length specified in the request, collect all data and construct RDMA-read response packet. If for any reason read could not be completed, master should send NACK as a response to the RDMA-read.
2. RDMA-write. Master should perform write operation of the PCI bus and send acknowledge cell back to the originator.
3. Other cycles (e.g. configuration) are treated similarly – each one is performed on the bus and acknowledge sent back to the originator.

The ACK/NACK payload for PCI master replies should follow NGIO Link spec, pp63-66

PCI slave – details

PCI slave is responsible to translate PCI cycles to NGIO cells and send them to the target on NGIO network. There are two ways to originate NGIO cell from the PCI:

1. Construct the cell explicitly in internal MT101 register and send to the fabric
2. Transform PCI cycle targeted to MT101 to NGIO cell.

The first option can be used for explicit (SW-visible) access of NGIO network (e.g. sending FMPs, initialization etc.).
PCI slave should have enough room to hold 32 posted writes. Reasoning: be able to run PCI bus write cycles back to back in MT101/102 system. Worst case would be 66Mhz PCI bus with 1-word writes back to back PCI writes. MT101 can run back to back writes at max throughput of 3 cycles (medium decode), e.g. MT101 accepts new cycle every 45 nSec. With best-case p2p latency of 700nSec, 16 slots for posted writes are enough. E.g 32 are good with margin.

FSA unit external definition

FSA unit (Funit) contains two key blocks – Fabric Service Agent (FSA) and Forward Data Base (FDB). FDB is responsible to translate MAC to port, including special cases handling (e.g. default port, squash port). FSA is responsible to accept all FMPs arrived to the device and either take appropriate action (if addressed) or forward FMP further. Note, that receive queues will forward all priority15 cells to FSA. FSA is also responsible to arbitrate CRBUS.

FDB

FDB contains a MAC to port translation table, which is mapped to MT101 control register space and is accessed through CRBUS.
FDB should implement the phase1 of data transfer protocol, delivering port number in a response to the MAC address. FDB also maintains the default port number (the one mapped to port number 255) and returns a physical port number if cell is routed to the default port. The port number is returned on lower 4 bits of ***FPORT*** bus in the phase1 of data transfer.
FDB also contains information about port speed. This information is returned on bits 4,5 of ***FPORT*** lines. It is used by receive queue to decide about cell buffering before its transfer to transmit queue.
In case MAC address translation requires cell discard (e.g. routed to port 254 or invalid MAC), FDB should return value of '1000000 on ***FPORT*** lines, and receive queue will discard the cell.

FSA

FSA unit has several functions:
1. Respond to FMPs arrived to MT101
2. Generate FMPs due to events generated inside MT101 and wait for acknowledge.
3. Implement System Port as specified.

Response to FMPs

FSA unit will accept all FMPs arrived to the MT101. It is mapped to internal port and should implement the internal data transfer protocol. FSA must have at least one buffer of full-size cell (292 bytes), where FMP will be placed upon arrival. FSA should decode the cell and act according to FMP content. The possible actions could be:
1. Forward cell to its destination – in case this FMP was not targeted to this device or its FSA
2. Execute read/writes to control registers of MT101 – in case this cell is FMPSet() or FMPGet(), addressed to this device. Construct – if needed – the reply FMP and send it to its destination. In case reply FMP cannot be constructed (COD requested is not supported in HW) – generate event (interrupt notification) and send it to Fabric Manager
3. Execute Direct Route protocol – upon arrival of Direct Route FMP.

Z-unit external definition and requirements

Z-unit is a unit that contains various miscellaneous functions. These functions do not necessarily form an overall bigger function (as other units), but rather grouped together for 'ease-of-management' purpose. Each sub-section of this chapter defines individual function of Z-unit

JTAG

JTAG unit is responsible to implement fully IEEE-compatible JTAG.

S-EPROM interface

Implements interface of MicroWire serial EPROM (spec is available in Data Sheets folder of the Outlook Public Folders) and implements all defined features.
S-EPROM unit (Sunit) is responsible to interface with S-EPROM. On power-up, this unit asserts *INIT* signal and starts reading the contents of S-EPROM and loads all control registers of MT101/102. The data format in EPROM is formed as a pair of 16-bit address and 32-bit data. Address with value of 0xffff means that all values to be written to the registers are read. After last control register is loaded, S-EPROM interface unit will clear *INIT* signal, so device will continue the boot process.
Sunit enables to program S-EPROM through *ROMDATA* and *ROMSTAT* register. This is 32-bit register that can be accessed by SW from PCI, FMP or CPU interfaces.
ROMDATA register contains 16-bit address and 16-bit data to be written to the ROM. ROMSTAT register is used to control the S-EPROM write operation:
Bit0 – write enable. After this bit is set, Sunit writes contents of ROMDATA[31:16] to address specified in ROMDATA[15:0]. This bit is cleared by HW after write has been completed.
Bit1 – write in progress. If set, it means the previous write command did not complete, and writes to ROMDATA register are ignored.
Bit2 – read enable. After this bit is set, Sunit reads contents (2 bytes) from the address specified in ROMDATA[15:0] and places it to ROMDATA[31:16]. This bit is cleared by HW after read has been completed
Bit3 – read in progress. If set, t means that previous read command did not complete and reads from ROMDATA register will return undefined data. Writes to ROMDATA register will be ignored

CPU interface

CPU interface unit (Cunit) is responsible to interface with auxiliary CPU that can (optionally) be attached to the MT101/102. CPU can implement local network management, therefore Cunit should enable CPU to access all relevant resources in the network. In particular, it provides hooks for:

1. Construct NGIO cell and send it to the fabric
2. Receive NGIO cell from the fabric
3. Access all configuration register of the device.

The first two functions are similar to what is provided by the PCI unit (see PCI spec for details). Access to internal registers is done by implicit addressing of these registers from the CPU interface (e.g. read will be interpret as control register read; write will be interpret as control register write).

Appendix B – reference designs (common blocks)

Timeout reference design

Limit counter

All timeout/limit counters in MT101 are loaded with initial (limit) value and counter is decrement each time event to be counted occur. The 'Timeout' event is generated when counter transitions from '1 to '0, and counter is re-loaded (if needed). This way loading ZERO to limit register will result in no-generation of the timeout/limit event.

Time clock

In compliance to NGIO standard, timeout limits are specified and counted in units of micro-seconds. The timeout counters clock – ***TCLK*** – will be generated inside the MT101 and all timeout limits in NGIO are specified in units of this clock. The clock is generating by dividing core clock by the value specified in the Time Divider register.

Appendix C – performanc

Latency analysis

The overall latency time between data receive and transmit is divided to three periods:
Period1 – time needed by the receive unit to get data from pins and place request for data transfer on internal bus
Period2 – time needed to arbitrate and deliver data over the internal bus network. Refer to the Data transfer summary section
Period3 – time needed by transmit queue to start data transfer on the pins from the clock it got the first data item of the cell.
For latency performance analysis it is assumed that there is only one transfer in the system between receive and target port.

Inter-unit protocol

Inter-unit protocol covers *Period2* of the data transfer. This period starts when MAC being extracted from the cell header, and is covered in Table 7

Stage	Best	Typical	Worst	Note
MAC to PORT	FAT	PN/2+FAT	PN+FAT	Zero for PCI-originated transactions (port is in the channel descriptor). For non-PCI FDB arbitration is overlapped with MAC extraction
Post request to TxQ	1	PN/2	PN	PCI will have highest priority, it will always be a single clock cycle – best, typical, worst
Data request from TxQ to RxQ	0	nCLK*TCL/2	nCLK*LCL	Internal clocks for PCI – written to internal buffer
Data xfer from RxQ to TxQ	0	TRD	LRD	RxQ not always can start data transfer (alignment issues)

Table 7 - latency summary for inter-unit dta transfers

The best case assumes that path required to send the data through is empty, and no arbitration delays interference while accessing common resources
The typical case assumes half of the maximum delay while accessing common resource and it assumes that target transmit queue is in the middle of transferring the cell of typical size.
The worst case assumes maximum delay while accessing common reource and it assumes that transmit queue just started to transmit cell of maximum length
Parameters are summarized in Table 8

Parameter	Meaning	latency
nCLK	NGIO port external clock	
FAT	FDB Access Time	3 internal cycles
PN	Number of ports (10 in MT101)	
TCL	Typical cell length	??+3 NGIO cycles
LCL	Largest cell size (292 bytes)	292/2+3 NGIO cycles
TRD	Typical RxQ delay from data request to data transmit	2 internal cycles
LRD	Longest RxQ delay from data request to data transmit	4 internal cycles

Table 8 - latency parameters

PCI to PCI bridge latency

The P2P latency is defined from the cycle FRAME# was asserted on the primary PCI bus till the cycle data is ready to be returned in the MT101 internal buffer. MT101 uses medium decode, which implies 2 PCI cycles to decode the address and assign channel.

MT101 architecture specification

Rev0.7

Michael Kagan
Mellanox Technologies

Table of content

System Archit cture Overview

System block diagram

MT101 is an NGIO switch element, with one of its ports being PCI. MT101 architecture enables system designer to build high-performance I/O system, capitalizing on advanced features of NGIO protocol (such as channel priority, reliability etc.) while using legacy I/O devices with PCI interface. The high-level system block diagram is shown on Figure 1.

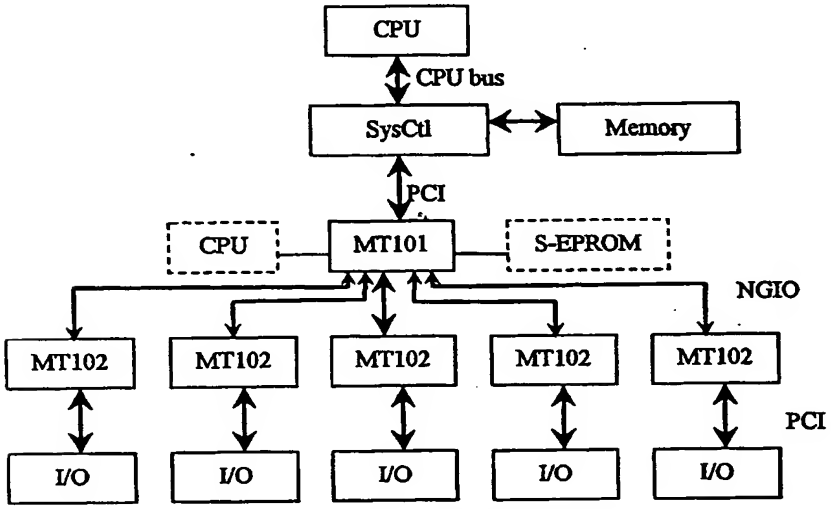


Figure 1 - MT101 system

PCI port supports 32 and 64-bit PCI bus, 33Mhz and 66Mhz. PCI-X bus support is being considered, but at this point it is out of scope of this document.
MT101 can be viewed from system PCI bus as a combination of P2P bridges and PCI to NGIO pridges, depending on the PCI function headers. Refer to P2P bridge configuration and boot section for details.

SW/HW architecture

The MT101/102 system provides a way to extend PCI-based system and utilize higher bandwidth by de-coupling different I/O devices, providing concurrent data transfer channels with higher bandwidth and priority-based queuing. The system contains of end-point agents (PCI unit, 8-bit CPU unit) and NGIO fabric, compliant to NGIO spec. Cell/packet sent to NGIO fabric can be squashed/corrupted inside the fabric without notification. End-points (e.g. PCI unit) must assure that data transfers are completed, and issue error (interrupt) message to SW in case cell got lost in the fabric or other error occurred.
Fabric management packets can be lost in NGIO fabric, and it is solely SW responsibility to assure FMPs arrival to their destination.
The interface between NGIO world and PCI world is implemented through two basic mechanisms:

1. Explicit NGIO cell generation. MT101/102 provides 292-byte storage and control/status register that will be used by SW to construct explicitly NGIO cell and send it to the fabric. This method will mainly be used by (but not limited to) initialization SW to construct messages to be sent to the fabric.
2. Implicit translation of PCI cycles to NGIO cells. MT101 will translate PCI cycles and events that need to be transferred to NGIO network, and schedule the cells/packets for transmission.

MT101 architecture also provides option to generate and forward interrupts caused by errors in NGIO fabric. Interrupts are forwarded to Master Fabric Manager through FMPs mechanism.

Interrupts' s handling

Under certain conditions MT101/102 can generate event to be delivered to SW. Examples could be link failure, interrupt or failure on secondary PCI bus etc. FMPs are used to deliver events to SW.
Once exception occurs on the device, the respective bit in cause register is set, and if not masked – FMP is sent to Fabric Manager containing cause register in its data payload. In response to this FMP, SW will read the cause register and clear the bits. FMPSet() is used to read and clear the cause register. The FMPSet() will contain mask with bits to be cleared in cause register. Implicit FMPGet() will return the cause register after bits were cleared.
In order to assure event delivery, FMPs are used to send the message. Since FMPs can get lost in the fabric, they will be re-sent after pre-defined timeout (if not acknowledged) and therefore multiple messages can be generated by both sides (HW and SW). In order to assure correct behavior regardless possible SW/HW races and possible loss of FMPs in fabric, following steps should be followed:

1. HW issues FMP send to event MAC, containing cause register as a data payload.
2. If within pre-defined period (programmable) cause register is not cleared by SW, FMP message is re-sent. If no response arrived, HW will cease generating messages (severe system problem, that will be discovered and taken care of during fabric sweep).
3. Upon event delivery to SW, response FMPSet() packet should be constructed and sent to the signaling device. Cause register should be addressed and data payload should contain the value of Cause Register reported. This will clear respective bits in the cause register and implied FMPGet()will return a new value of Cause Register. If returned value is not zero, it means that other interrupts arrived since original FMP generated, and SW must issue another FMPSet() until zero value is returned.
4. Only after cause register is cleared, the interrupt handling routine can start.

Errors reporting

In addition to the event reporting mechanism through FPMs and interrupts, MT101 provides HW hooks that allow monitoring of the events being collected inside the device. All cause registers of MT101 have shadow shift registers associated with them and these shift registers are connected to the chain. This chain is loaded first time after RESET dis-asserted and scanned out to the SDO pin. After scan is completed, the shift register is loaded again and shifted again, hereby providing real-time information about errors registered in the device. SCLK is a clock of scan-out and SSTRB indicates the beginning of the chain. The order of the registers in the chain is:

1. Consolidated Cause Register
2. PCI port cause register
3. NGIO0 port cause register
4. NGIO1 port cause register
5. NGIO2 port cause register
6. NGIO3 port cause register
7. NGIO4 port cause register
8. NGIO5 port cause register
9. NGIO6 port cause register
10. NGIO7 port cause register

External logic can decode this information and raise the flag (e.g. light up LED) if certain error ocucred in the system.

Data integrity

Internal data integrity

Data integrity in MT101/102 devices is assured by validating CRC in both input and output of the device. Upon receive of the cell, CRC is calculated and validated against CRC field in the cell. If mismatch encountered and end-of-cell delimiter is not EP, the *receive_error* counter of respective port is incremented. If cell transmission has not been started when error encountered, the cell will be discarded inside the MT101. If cell transmission has already been started, it will be transmitted with EP end-of-cell delimiter.
While transmitting the cell, CRC is validated again in the transmit queue. If CRC error encountered in the transmit queue and no error indication received from the receive queue, it means the cell was corrupted

inside the MT101. In this case *internal_error* counter for respective transmit queue will be incremented and EP delimiter will be attached to the cell.

PCI errors handling

If MT101 encounters data parity error during the PCI cycle, it reports parity error as specified in PCI bus spec. In addition, the cell generated will be completed with EP delimiter and *PciError* counter will be incremented. If cell has not been sent to the NGIO fabric, it will be discarded. PCI target unit will not wait for acknowledge for such a cell.
Transmit queue attached to the PCI units (master and target) validate cells' correctness (CRC and error delimiter) before transferring cycle to the PCI bus. Corrupted cells are dropped, and error counter is incremented.
Under certain conditions PCI slave can deliver corrupted data to the PCI bus. This happens when PCI read re-try occurs right after its response arrives to the PCI slave, and data is being bypassed to the PCI bus before end of cell (delimiter, CRC etc) has arrived and cell correctness was validated (CRC, delimiter). In this case PCI unit will increment *PciError* counter. If cycle was not yet completed by the master when CRC error was encountered, PCI unit will abort the cycle (target abort)[1].
Certain errors can be caused by erroneous configuration of the PCI port. These errors will be logged and (optionally) reported through interrupt or SERR mechanism.

NGIO errors handling

Each receive port contains a counter that counts corrupted cells arrived to the port. Cells with EP delimiter are not counted. If error encountered before cell transmit starts, it will be squashed in the MT101/102

Error reporting

All error counters of MT101 can generate interrupt to the fabric manager. If value of the counter reaches respective limit value, interrupt is generated, unless it is masked in the Cause Mask Register. Setting limit to zero disables interrupts.

Minimizing errors in the network

In order to minimize flow of erroneous messages in the NGIO fabric, each receive port of MT101 should be programmed to buffer entire cell before its forwarding to the destination queue. Note that in such a mode the latency of the communication will increase and overall bandwidth utilization will be lower. However, each receive queue will have a chance to examine cell for correctness (CRC, delimiter) before scheduling its transmission, and erroneous cells will be squashed. Although this mode is not recommended for mainstream operation, it can be handy for system debug and searching for unreliable links.

Access ordering and fences

Support for ordering and fences in MT101 system is equivalent to those of NGIO. Cycles' ordering is preserved only within the same channel. Fence can be implemented on a single channel only (not on entire system). Hence, in MT101/102 system support for fence barriers originated from PCI is limited to a single channel the fence was issued to. In other words, fence will work correctly if communication path between fencing and fenced device is limited to single priority and each device has only one MAC and WQPN assigned to it.

NGIO priorities

MT101/102 resource management supports 4 priorities in HW. Eight NGIO priorities (zero to 7) are mapped to four HW-supported priorities as defined in NGIO spec.

LiveLock

MT101 provides capability to prevent LiveLock (when high-priority traffic blocks entirely lower-priority one). This option is provided through LiveLock register, defined for each one of the four priority queues in

[1] Note that it is not guaranteed that bogus cell will be completed with target abort, as bus master may complete the cycle (de-assert ***FRAME#***) before tail of cell arrived to CRC checker. In order to assure that no bogus data is delivered to PCI bus, MT101 should be configured as 'store and forward' – e.g. receive queue should stack up the data before forwarding to the PCI port.

each transmit queue. After queue transmitted number of cells pre-set in its associated LiveLock value, it 'gives up' a link for lower-priority queue for a single cell transfer. If no cells are ready f r transmission in lower-priority queue, it will decrement its counter without transmission. The slot can be further 'given up' to even lower priority queue under same conditions.
Setting LiveLock value to zero disables the mechanism (e.g. cells will be transmitted in strict priority order).

Flow control

MT101 may issue flow control due to resource' overflow – either data array in the receive port is filling up or port is running out of descriptors (pointers) for arriving cells. The initial version of MT101 includes 2Kbyte data buffer and 16 pointers per each receive port.
Flow control thresholds are configurable through a pair of FC configuration registers – one for data-driven flow control and another for pointers-driven. The each priority has a resource watermark associated with it, and once availability of the resource goes below the watermark, respective flow control is issued. The watermarks are programmed per NGIO cell priority. Since MT101 implements only four priority queues, flow control will be issued in accordance to MT101 priority queues – e.g. if NGIO priorities 3,4 and 5 are bundled to the same MT101 priority queue and resource availability gone below priority 3, MT101 will issue flow control of priority 5 because it is the highest priority sharing same resource.
Data threshold is specified in 64-bit FCDataConfig register. Each byte defines the threshold of empty space in data array in 16-byte chunks for corresponding priority (e.g. byte0 corresponds to priority 0, byte 7 to priority 7).
Pointers' threshold is specified in 64-bit FCPointerConfig register, Each byte defines the threshold of empty pointers for corresponding priority (e.g. byte 0 corresponds to priority 0, byte 7 to priority 7)
Figure 2 shows flow control configuration header.

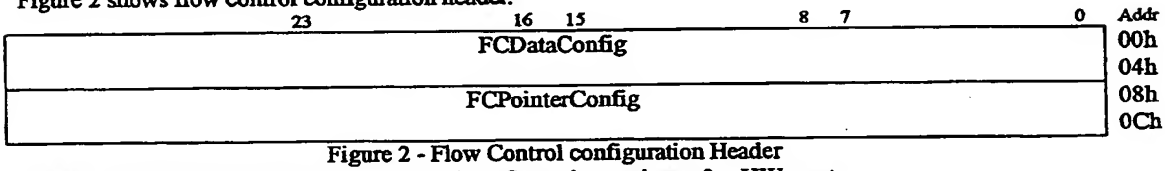


Figure 2 - Flow Control configuration Header

Table 1 Specifies the values of Flow Control configuration register after HW reset

Field	Value	Comment
FCDataConfig	2A2A3F3F55556B6Bh	Assures 2 full cells after XN8 + 4 chunks for sync
FCPointerConfig	1111151519191D1Dh	Keeps 17 pointers after XN8

Table 1 - Flow Control configuration register reset values

In order to avoid LiveLock at system level, flow control for priority P can be issued only if there is at least one cell of this priority waiting in the queue. This mechanism will prevent high-priority traffic fluctuating around its threshold to block entirely lower-priority traffic.

PCI to NGIO Interface

PCI cycles are converted to NGIO cells and sent to the fabric by PCI interface unit of MT101/102. Entire address space (memory and I/O) is divided into segments (channels), and each segment is mapped to NGIO channel (WQPs, priority, MAC etc.). Auxiliary attributes of the channel are used to determine length and type of transfer (e.g. prefetch depth). These attributes are configurable in SW through configuration registers of MT101.

Port speed match

MT101 enables to classify ports to four different speeds. Receive/transmit arbitration logic uses this information to buffer enough cell data in the queue before start the transmission to avoid overrun on one hand and start cell transmission as soon as possible on the other hand. Port speeds classified as Slow, Medium, Fast and Very Fast and their encoding defined in Table 2.

Port Speed	Encoding
Slow	00
Medium	01
Fast	10

Very Fast	11

Table 2 - Port speed encoding

Table 3 summarizes the rules for data buffering.
Px – partial buffering needed, Px% of the cell should be buffered

		Destination TxQ speed			
		Slow	Medium	Fast	VeryFast
RxQ speed	Slow	P1	P2	P3	P4
	Medium	P1	P1	P2	P3
	Fast	P1	P1	P1	P2
	VeryFast	P1	P1	P1	P1

Table 3 - Receive/transmit port speed relation

P1, P2, P3 and P4 are programmable through PortBuffer register, encoding is defined in Table 4.

Value	Buffering
00	Zero
01	½ cell
10	¾ cell
11	Full

Table 4 - Port Buffering programming

Zero means there is not buffering constrains, and cell can be transmitted immediately to its destination port.
Full means that full cell must be buffered in receive queue before transmission.

Serial EPROM – initialization

MT101/102 can be initialized from microwire S-EPROM. EPROM is programmed in chunks of 3 16-bit words. The first word contains address of the control register and subsequent two words contain the data to be written to the register. On power-up MT101 sequentially reads the EPROM and loads its internal registers. The last chunk is identified by FFFFh address and its data is ignored.
MT101 enables to program S-EPROM through *ROMDATA* and *ROMSTAT* registers. These registers can be accessed by SW from PCI, FMP or CPU interfaces.
ROMDATA register contains 16-bit address and 16-bit data to be written to the ROM. *ROMSTAT* register is used to control the S-EPROM write operation:
Bit0 – write enable. After this bit is set, contents of *ROMDATA*[15:0] is written to address specified in *ROMDATA*[31:16]. As long as this bit set, it means write has not been completed and writes to *ROMDATA* register are ignored. After write operation is completed, the bit is cleared by HW.
Bit1 – read enable. After this bit is set, contents (2 bytes) from the address specified in *ROMDATA*[31:16] is read and placed it to *ROMDATA*[15:0]. This bit is cleared by HW after read has been completed. Result of reading ROMDATA register while this bit is set is undefined. Figure 3 shows template of ROMDATA and ROMSTAT registers.

31 24	23 16	15 8	7 0	Addr
ROMDATA – address		ROMDATA – data		00h
ROMSTAT				04h
ROMCLK				08h

Figure 3 – S-EPROM control registers

ROMCLK register is used to divide system clock to generate clock input for serial EPROM. Internal clock is diviced ny the value stored in ROMCLK register and is used to generate S-EPROM clock.
Table 5 defines reset values of S-EPROM configruation reigster.

Field	Value	Comment
ROMDATA	0h	
ROMSTAT	0h	No 'external' ROM operation
ROMCLK	80h	Divide internal clock by 128 to generate ROM clock (max 2Mhz)

Table 5 - S-EPROM configuration register reset values

SERDES configuration

MT101 is capable to configure SERDES, using a subset of MII (Management Information Interface) as defined in IEEE 802.3. The configuration is compatible with AANetCom device as defined in its data

sheet. MT101 deploys SERDES configuration registers – *SDATA* and *SSTAT* – for this purpose. *SDATA[15:0]* contains 16-bit data to be written to the SERDES, *SDATA[20:16]* contains address of internal SERDES register and *SDATA[28:24]* selects SERDES device to be accessed. SSTAT register is used to control the SERDES access operation:

Bit0 – write enable. After this bit is set, contents of *SDATA[15:0]* is written to register whose address specified in *SDATA[20:16]* in device specified in *SDATA[28:24]*. As long as this bit set, it means write has not been completed and writes to *SDATA* register ignored. After write operation is completed, the bit is cleared by HW.

Bit1 – read enable. After this bit is set, contents (2 bytes) from the SERDES register specified in *SDATA*[20:16] of device specified in *SDATA*[28:24] is read and placed it to *SDATA*[31:16]. This bit is cleared by HW after read has been completed. Result of reading *SDATA* register while this bit is set is undefined. Figure 4 shows template of *SDATA* and *SSTAT* registers.

31 29	28 24	23 21	20 16	15 8	7 0	Addr
reserved	Device#	reserved	Reg. addr	SDATA – data		00h
SSTAT						04h
SERCLK						08h

Figure 4 - SERDES configuration registers

Table 6 shows SERDES configuration values after HW reset.

Field	Value	Comment
SDATA-data	0h	
Reg. addr	0h	
Device#	0h	
SSTAT	0h	
SERCLK	80h	Divide internal clock by 128 to generate MDC (max freq of MDC is not clear

Table 6 - SERDES configuration reset values

PCI/NGIO system architecture

Initialization

MT101 can be initialized from HW reset, and each unit can be reset separately through *SoftReset* register. Each bit in this register asserts HW reset for different units in the device, as specified in Table 7.

Bit	Unit
0	*INIT_PCI* signal is asserted
1	*INIT_NGIO* signal is asserted
2	PCI Target
3	PCI Master
4	NGIO Port0
5	NGIO Port1
6	NGIO Port2
7	NGIO Port3
8	NGIO Port4
9	NGIO Port5
10	NGIO Port6
11	NGIO Port7
12	FSA NGIO port
13	PCI NGIO port
14-31	reserved

Table 7 - SoftReset register

All bits of SoftReset registers are set at reset and cleared during the second phase of MT101 initialization. There are three phases of MT101 initialization:

Phase1 – HW reset.

After HW reset MT101 wakes up as a switch with all configuration registers loaded with their default. *INIT* signal is asserted[2]. Phase1 is completed when HW reset is de-asserted.

Phase2 – S-EPROM sequence.

S-EPROM interface unit loads configuration registers with new values (if needed). All, some or none registers can be loaded in this phase. The first step should be to clear respective bits in the SoftReset register. All interfaces to the external world should be ignored until *INIT* signal is cleared. All NGIO links should be down, all PCI cycles should be delayed, all CPU cycles should be delayed. Upon completion of this phase the device is ready to operate.

Receiving FFFFh as a control register address from S-EPROM interface is an ultimate completion of second initialization phase. If no S-EPROM is present, the data lines should be tied to '1, so FFFFh address will be read by the HW on the first access to S-EPROM

MT101 implements Timer, shown Figure 5

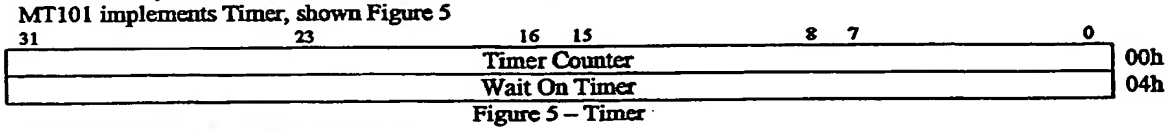


Figure 5 – Timer

Timer is cleared (zero) after HW reset

Timer is implemented through two registers – Timer Counter and Wait On Timer register. Timer Counter is a read/write register, and it counts internal clocks of MT101, incrementing value of Timer Counter every clock.

Wait On Timer register implements a Wait function. Write to this register will not be acknowledged by HW till value of Timer Counter does not match the value written to Wait On Timer register. This

[2] There are two separate *INIT* signals – *INIT_PCI* and *INIT_NGIO*, so PCI and NGIO ports are not tied up to start together. This can be used for S-EPROM-driven system initialization, when NGIO traffic should start before PCI side wakes up.

functionality is enabled only while accessing Wait On Timer register from S-EPROM unit only. Timer register is handy while booting entire system form S-EPROM, and is used to implement equivalent of spin-wait loops in the CPU.
The first two phases of boot will be called 'embedded configuration' in future references.

Phase3 – External initialization.

After *INIT* signal is cleared , each NGIO port brings link up and PCI port starts accepting PCI cycles. External world can change configuration set in phase1 and/or phase2 using FMPs and PCI configuration cycles.
MT101 can be configured for different combinations of P2P bridges and/or multiple PCI to NGIO (P2N) bridges, as specified in Table 8. Therefore MT101 SW initialization should contain two parts, corresponding to P2P and P2N boot/configuration.

Configuration number	Number Of P2Ps	Number Of P2Ns	Total number of functions
0	0	8	8
1	1	0	1
2	1	1	2
3	1	2	3
4	1	3	4
5	1	4	5
6	1	5	6
7	1	6	7
8	1	7	8
9	2	0	2
10	2	1	3
10	2	2	4
12	2	3	5
13	2	4	6
14	2	5	7
15	2	6	8
16	3	0	3
17	3	1	4
18	3	2	5
19	3	3	6
20	3	4	7
21	3	5	8
22	4	0	4
23	4	1	5
24	4	2	6
25	4	3	7
26	4	4	8
27	5	0	5
28	5	1	6
29	5	2	7
30	5	3	8
31	6	0	6
32	6	1	7
33	6	2	8
34	7	0	7
35	7	1	8
36	8	0	8

Table 8 - MT101 configuration combinations

In case where total number of functions is less than maximum possible (e.g. 8), it means that multiple NGIO ports are bundled to implement 'virtual' PCI bus.
Each function implements full configuration space header as defined in PCI spec. There are total eight function configuration templates in MT101, which are initialized during embedded configuration phase. P2P function will use P2P header type. The format of P2P header is defined in the P2P Bridge Architecture spec, and format of PCI device header is defined in PCI architecture spec. P2N function will use type 00h header, and its configuration fields are defined in Table 9. P2P function uses type 01h header, and its fields defined in Table 10.

Field	Value	Comment
VendorID	MLNX	Mellanox, to be defined
DeviceID	MT101	to be defined, RO from PCI
Command	Programmed	Cannot set special cycle bit
Status	Programmed	Capabilities list bit is cleared
RevisionID	MT101	Extension to deviceID
Class Code		RO from PCI
Cache Line Size	Programmed	
Latency Timer	Programmed	Function per spec
Header Type	00h, 80h	Set by S-EPROM
BIST	Per spec	
BAR	Programmed	16 least-significant bits are zero
Cardbus CIS Pointer	ZERO	Not implemented
Subsystem Vendor ID	Per spec	S-EPROM
Subsystem ID	Per spec	S-EPROM
Expansion ROM Base Address	Zero	Not implemented
Capabilities Pointer	Zero	Not implemented
Interrupt Line	Programmed	
Interrupt Pin	Programmed	
Min-Gnt	Programmed	
Max-Lat	Programmed	

Table 9 - Type 00h (P2N) configuration header - fields definition

Programmable fields are cleared (zero) by HW reset and set by the S-EPROM or configuration SW

Field	Value	Comment
VendorID	MLNX	Mellanox, to be defined
DeviceID	MT101	to be defined, RO from PCI
Command	Programmed	Cannot set special cycle and VGA palette snoop bits
Status	Programmed	Capabilities list bit is cleared
RevisionID	MT101	Extension to deviceID
Class Code	P2P	P2P, RO from PCI
Cacheline Size	Programmed	
Primary latency timer	Programmed	
Header type	01h, 81h	Set by S-EPROM
BIST	Per spec	
BAR	Programmed	16 least-significant bits are zero
Primary bus number	Programmed	
Secondary bus number	Programmed	
Subordinate bus number	Programmed	
Secondary latency timer	Zero	Not implemented
I/O base	Programmed	Implemented through segments
I/O limit	Programmed	Implemented through segments
Secondary status	Zero	Not implemented
Memory base	Programmed	Implemented through segments
Memory limit	Programmed	Implemented through segments
Prefetchable memory base	Programmed	Implemented through segments
Prefetchable memory limit	Programmed	Implemented through segments

Field	Value	Comment
Prefetchable base upper 32 bits	Programmed	Implemented through segments
I/O base upped 16 bits	Programmed	Implemented through segments
I/O limit upper 16 bits	Programmed	Implemented through segments
Capabilities pointer	Zero	Not implemented
Expansion ROM base address	Zero	Not implemented
Interrupt line	Programmed	
Interrupt pin	Programmed	
Bridge control	Zero	Not implemented

Table 10 - Type 01h (P2P) configuration header - fields definition

Programmable fields are cleared (zero) but HW reset and set by S-EPROM or configuration SW.

NGIO channels configuration for PCI

PCI/NGIO interface is configurable through programming Channel Headers – Target Channel Header (PCI target) and Master Channel Header (PCI master). Channel Headers are programmed as a control registers of the MT101/102. They can be programmed directly from PCI interface (on MT101), by issuing FMPSet() operations (MT102), from serial EPROM or from attached 8-bit CPU.

Channel Headers contain all information about the NGIO channel and mapping between NGIO channel and PCI cycles' space. Each Channel Header contains address (BAR and Limit) that is mapped to this channel, type of the cycle (I/O, memory, configuration) and defines all NGIO cannel attributes (MAC, WQPN etc.) MT101 will claim cycle from PCI bus based on lookup in the Channel Headers and construct NGIO packet accordingly; BAR registers in PCI Function headers are ignored except for the first one, which is used to access internal registers of MT101.

NGIO channels configuration on PCI target

PCI target unit contains 32 PCI Target Channel Headers. Each Channel Header represents NGIO WQP. The WQP number is constructed by appending serial number (from 0 to 31) of Channel Header to upper 9 bits of *TargetWqpBase* register.

The Target Channel Header format is presented in Figure 6.

31 24	23 16	15 8	7 0	Addr
BAR		Address Map	Map Mask	00h
BAR – upper 32 bits				04h
Limit		RESERVED		08h
Limit – upper 32 bits				0Ch
RESERVED	Cache line size		Prefetch length	10h
Channel type	DR pend / DR cap	PSN – outbound	Port num. / Priority	14h
Remote WQPN		Destination (remote) MAC		18h
MH (Memory Handler)				1Ch

Figure 6 – PCI Target Channel Header format

BAR, Limit fields define address space segment, as specified in PCI to PCI bridge spec.

The segment validity is implicit – if BAR and LIMIT have same values, segment is invalid.

Address Map along with Map Mask is used to re-map the most significant bits of the original address. The upper 8 bits of the new address are constructed by implementing following operation on upper 8 bits[3] of the address:

NEW_ADDRESS = (OLD_ADDRESS and NOT(MAP_MASK)) or ADDRESS_MAP

Source MAC for the NGIO cell is taken from the PCI MAC address register (single MAC for PCI port).

Channel type field defines channel characteristics and is shown on Figure 7.

[3] bits 63:56 for 64-bit address and bits 31:24 for 32-bit address

7 5	4 3	2 0
Reserved	Channel Type '00 – non-connected, not acknowledged '10 – connected, not acknowledged '11 – connected, acknowledged '01 – reserved	Cycle type '000 – prefetch memory '001 – non-prefetch memory '010 – I/O '011 – reserved '100 – configuration type0 '101 – configuration type1 '11x – reserved

Figure 7 - Channel type

The channel type of must be programmed to connected acknowledged channel. PCI unit does not support other channels' types.

The cycle type information is used by PCI target to construct NGIO cell, and it is used by master to issue correct command on C/BE# lines of the cycle. NGIO cells arrived to memory channel will be issued as memory read/writes. NGIO cells arrived to I/O channel will be issued as I/O reads/writes. NGIO cells arrived to configuration channel will be issued as configuration read/writes.

PSN-inbonud and CSN-inbound fileds store the respective expected serial number. PSN-outbound field stores the PSN number that will be used for next packet generated from the channel.

DR Cap field define number of oustanding delayed request packets that can be handled by the other side of the channel. Minimum number is four[4].

DR Pending field is initialized to zero, incremented each time new Delayed Request packet is sent to the channel and decrement each time Delayed Request packed is removed from the wait queue (e.g. acknowledged, NACK'ed or timeout expired). If value of DR Pending matches value of DR capacity, no new delayed request packets are allowed to be sent to the channel till enough outstanding delayed requests will be acknowledged. If DR capacity field is zero, it means unlimited capacity of the far end of the channel.

Prefetch length field is used to determine depth of prefetch (RDMA-read length) for read cycles that require more than a single PCI bus transfer (FRAME# is asserted for more than one cycle, Memory Read Multiple, Memory Read Line cycles). If this field is zero, no prefetch should be done (single-transfer cycle). The prefetch length is specified in bytes.

Table 11 defines the value of PCI Target Channel Configuration Header after HW reset.

Field	Value	Comment
BAR	0	
Limit	0	
Prefetch Length	0	
Cache line size	0	
Map Mask	0h	To assure no address re-map by default + easy BAR programming
Address Map	0	To assure no address re-map by default + easy BAR programming
Channel type	0	
PSN	0	
DR Cap	4	No more than 4 delayed requests allowed on the channel
DR pend	0	
Priority	0	
Port num	0	
Remote MAC	FFFFh	Permissive MAC address
Remote WQPN	0	FMP queue
MH	0	

[4] Each outstanding delayed request consumes one slot in the channel capacity. Single PCI read can generate up to 4 delayed requests (if byte enables are alternating), e.g. single PCI cycle can consume up to four slots of channel capacity. Hence PCI target will deny (re-try without generating delayed request) any PCI read if remaining channel capacity is less than four.

Table 11 - PCI Target Channel Configuration header reset values

NGIO channels configuration on PCI master

PCI master unit contains 32 PCI Master Headers, each one representing NGIO WQP. The WQP number is constructed by appending serial number (from 0 to 31) of Channel Header to upper 11 bits of *MasterWqpBase* register. The Master Channel Header format is presented in Figure 8.

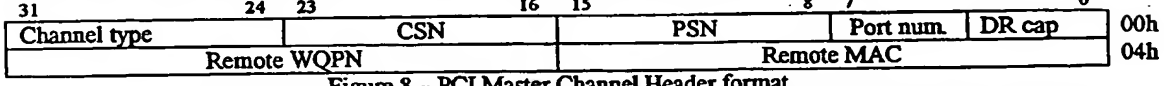


Figure 8 – PCI Master Channel Header format

DR Cap field defines number of inbound request buffers allocated for this channel. For delayed writes data payload length should not exceed 8 bytes.
PSN and CSN fields store next packet/cell number expected to arrive on the channel
All fields of PCI Master Channel Header are cleared (zero) at HW reset.

P2P bridge configuration and boot

The goal of P2P configuration algorithm is to make it as close as possible to 'native' PCI initialization, with MT101-specific code encapsulated.
In order to configure MT101 as a P2P with (optional) multiple NGIO links bundled to implement a single 'virtual' PCI bus, SW needs to implement steps summarized in Table 12. The table outlines which steps are implemented during 'native' P2P initialization and can be executed without SW modifications and which steps require MT101-specific code.

Step	Function	Comments
1	Set BAR values	Standard SW – sweep PCI bus, read configuration registers, set BAR value for accessing the internal registers.
2	Establish channels for MT101/102 configuration	MT101-specific code. Establish channels between MT101 PCI port and all MT102 PCI ports (assign MAC addresses, WQPs etc). Establishing the channels requires access to MT101 internal registers, which can be done from PCI interface for general fabric configuration. This step can be avoided by programming entire NGIO fabric from S-EPROM of MT101.
3	Complete 'standard' system initialization	Standard SW – sweep entire system, assign secondary busses numbers, assign BARs to all devices in the system, assign address space mapping (base and limits) in all P2P segments.
4	Reflect configuration parameters and address mapping to all MT101/102 devices in the system	MT101-specific code. Establishes segment (channels) in each MT101 and MT102 by configuring channels in MT101/102 internal configuration registers.

Table 12 - P2P initialization steps

Note that – as any PCI configuration – the configuration process is recursive. If during system sweep in second step configuration SW discovers MT101 device on secondary PCI bus(es), it has to implement step1 and step 2 over again for that bus.
Secondary P2P bus can reside behind single NGIO port or can be spread between number of NGIO ports. During the third step of PCI configuration, MT101 needs to have all routing information in order to route type1 configuration cycles to the right destination. This information is provided through *PciPortConfig* registers. There are total of eight registers (one per each NGIO port), and their template is shown in Figure 9.

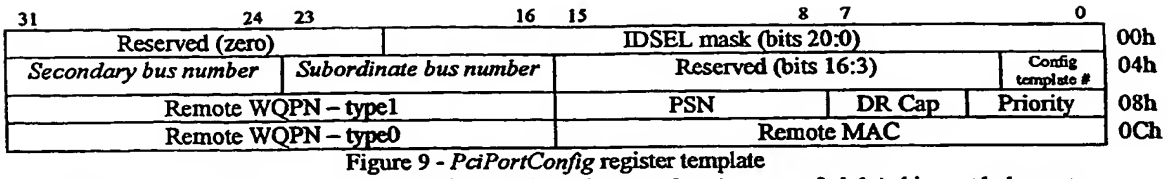


Figure 9 - *PciPortConfig* register template

Configuration template number defines which configuration template (one out of eight) this port belongs to.
IDSEL mask field is an OR of decoded device numbers (IDSEL) of the secondary bus devices that are mapped to this NGIO port[5], and it is limited to 21 device.
Fields specified in italic are alias from the configuration template, defined by configuration template number field in the register[6].
When PCI unit of MT101 observes the configuration type1 cycle on PCI bus, it looks up the *PciPortConfig* registers to identify whether this cycle should be claimed by MT101. If Bus Number field of the type1 cycle belongs to the range covered by MT101 (e.g. it fails in one of the secondary bus ranges covered by MT101 functions), it claims the cycle.
If Bus Number field equals to one of the secondary bus numbers covered by MT101, the cycle is converted to type0 configuration cycle, and NGIO RDMA cell constructed. The destination port is one whose Secondary Bus Number filed in *PciPortConfig* register matches Bus Number field of original type1 transaction, and decoded value of Device Number field in original PCI cycle is not masked (nullified) by IDSEL Mask value in *PciPortConfig* register.
If Bus Number field in type1 transaction belongs to the range covered by MT101, but not equal to any of its secondary bus numbers, MT101 generates NGIO RDMA cell with type1 configuration. Destination is determined from *PciPortConfig* registers using Secondary Bus Number and Subordinate Bus Number fields.
The resulting cell is sent to the channel whose number is specified in *Type0 Channel #* field of the *PciPortConfig* register for type0 configuration cell, and to *Type1 Channel #* field for type1 configuration cells. The segment (channel) registers on both sides of the channel should be programmed appropriately to assure correct operation.

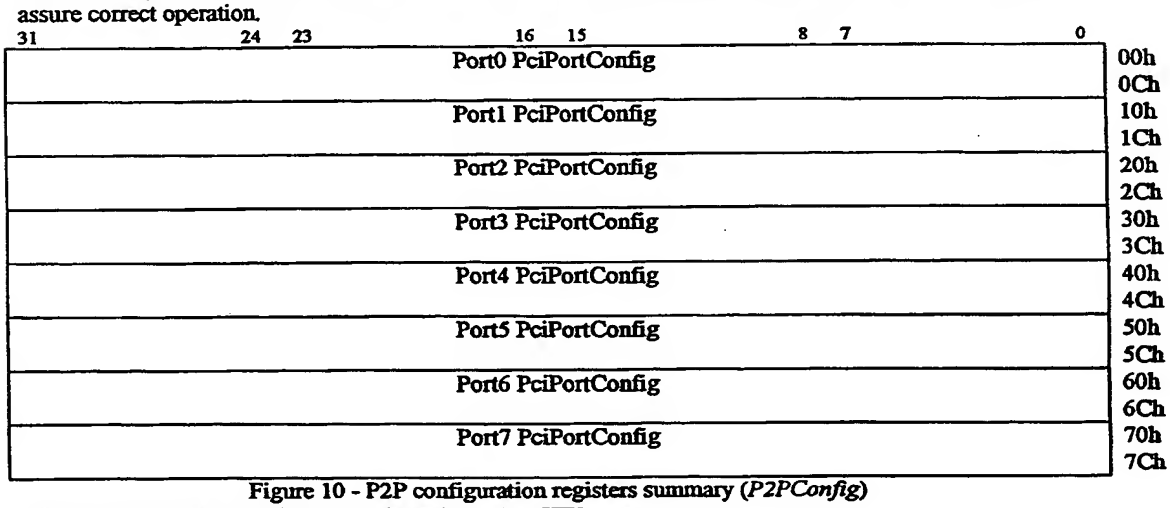


Figure 10 - P2P configuration registers summary (*P2PConfig*)

All P2P configuration registers are cleared (zero) at HW reset.

---

[5] If secondary PCI bus is mapped to a single NGIO port this register corresponds to, all bits should be set in IDSEL field

[6] Note that IDSEL mask and Configuration Template Number fields in *PciPortConfig* register are filled in during embedded configuration. The second step in PCI configuration (Table 12) is necessary in order to assign MAC addresses for PCI masters in MT102s and fill in MAC Address field in *PciPortConfig* registers. If MAC addresses can be assigned during embedded configuration phase, the second step can be skipped and PCI initialization SW can run without interception.

Figure 11 illustrated assignment of WQP numbers in PCI target unit.

	15 ... 6	5	4	3 ... 1	0
Addr. segment	Target WQPBase	0	Segment header #		
Config, type0	Target WQPBase	0	0	PciPortConfig #	0
Config, type1	Target WQPBase	0	0	PciPortConfig #	1

Figure 11 - PCI target unit WQPs assignment

PCI/NGIO interface

PCI cycles conversion to NGIO cells

After all channels are configured as specified in previous sections, MT101/102 can automatically convert PCI cycles to NGIO cells and send them to the NGIO fabric. In addition, NGIO cells that arrive to PCI destination channels will be automatically converted to PCI cycles.

Once PCI slave decoded PCI cycle that maps to its address space, it converts it to the NGIO cell obeying NGIO rules. Table 13 summarizes PCI CMD to NGIO translation. PCI unit issues only RDMA-read and RDMA-write cells to the NGIO fabric. For PCI destinations, the C/BE# of the PCI cycle will be determined by the PCI master based on channels attributes cell arrived to.

CMD	Command	NGIO cell
0000	INTA	None
0001	Special cycle	None
0010	I/O Read	RDMA-read, length as specified in original cycle
0011	I/O Write	RDMA-write, length as specified in original cycle
0100	Reserved	None
0101	Reserved	None
0110	Memory Read	RDMA-read, length according to Prefetch Length field in Target Channel Header format (if prefetch memory)
0111	Memory Write	RDMA-write, length as specified in original cycle
1000	Reserved	None
1001	Reserved	None
1010	Configuration Read	RDMA-read, length – depending on BEs, 4 bytes at most
1011	Configuration Write	RDMA-write, length – depending on BEs, 4 bytes at most
1100	Memory Read Multiple	RDMA-read, length according to Prefetch Length field in the Target Channel Header (if prefetch memory)
1101	Dual Address Cycle	None
1110	Memory Read Line	RDMA-read, length according to Prefetch Length field in the Target Channel Header (if prefetch memory)
1111	Memory Write and Invalidate	RDMA-write

Table 13 - PCI cycle CMD to NGIO cell translation

Cell will be legal NGIO cell:

1. MAC, port number, priority, PSN, MTH and WQ pair (source and destination) are taken from respective channel (BAR segment).
2. The NGIO data access must be consecutive string of bytes. If not all *BE#* signals were asserted in the PCI cycle, slave must split this cycle to multiple NGIO cells and assure that each one contains consecutive byte string (read or writes)

Following are the rules for RDMA-read cells generation:

1. For read-multiple PCI cycles slave generates single cell.
2. For read-multiple PCI cycles length of RDMA-read is taken from configuration register associated with the channel the read is targeted to.
3. Length of the read must obey PCI rules for data prefetch.
4. RDMA-read request should never cross 32-bit and 64-bit address boundary (e.g. start address + length should never wrap around 32 or 64-bit address).

If RDMA-read response does not arrive within time period set in MemLifeTime register, Read Reply timeout counter is incremented and read request is removed from the arrival queue. If Read Reply Timeout limit exceeded, MT101 issues Target Abort for the next retry of this read and removes transaction from the pending transactions queue. This mechanism enables to re-transmit read requests that got lost in the fabric.

RDMA-write generation rules:

1. Length or multiple writes is limited to 128 bytes (? Maybe just unlimited? Just based on the buffer availability? What about alignment?)
2. For posted writes TRDY# is returned immediately and RDMA-write is sent to the target. If not acknowledged within time specified in MemLifeTime register, interrupt is generated.
3. For non-posted writes, cycle is stopped (re-try), original data is kept and RDMA-write cell sent to the NGIO network. When write originator re-issues the cycle after this cell was acknowledged, PCI slave compares all cycle attributes (address, data, byte enables etc.) with original cycle, and if match occurred – TRDY# is returned to the originator.

If RDMA-Write is not acknowledge times out, it is treated same way as RDMA-read.

NGIO cells conversion to PCI cycles; acknowledges

NGIO cells that arrive to the PCI unit will be converted to the PCI cycles. The command driven on C/BE# lines of the PCI bus will be in accordance to NGIO cell type and channels attributes specified in Channel Type field of the Master Channel Header.

Cycle Type	RDMA-read	RDMA-write
'00, (prefetch memory)	'0110 or '1100 (depending on length)	'0111
'01 (non-prefetch memory)	'0110	'0111
'10 (I/O)	'0010	'0011
'11 (configuration)	'1010	'1011

Table 14 - NGIO cells to PCI cycles translation

PCI master channels will normally be configured to connected acknowledged service, and acknowledge will be generated for each packet arrived. If PCI bus operation cannot be completed, NACK will be sent back to the requestor. Table 15 defines NACK payload in case of PCI operation cannot be completed and its respective action on PCI target (requestor) side

NACK payload	Reason (master)	Action (requestor)
1 (sequential error)	PSN, CSN mismatch	Remove request from wait queue, increment sequence error counter, issue target abort. Reset PSN to NACK'ed PSN number, so next request will have same PSN number as one that was NACK'ed.
2 (out of bound error)		
3 (remote access error)	Target or master abort, parity error, wrap around 32 or 64-bit address	Issue target abort on cycle retry; increment respective error counter
4 (catastrophic error)		
5 (operation error)		

Table 15 - PCI NACK reply payload

PCI cycles generation from NGIO interface

MT101 architecture provides a way to generate cycles on PCI bus through programming *PciSpecialCycles* registers. These registers are accessible through FMPSet() operation, hereby enabling generate PCI special cycles from NGIO interface. Every PCI cycle can be generated through this mechanism. Data transfer length is limited to 8 bytes.

The mechanism is provided through *PciCycle* control register. Figure 12 illustrates its format and fields.

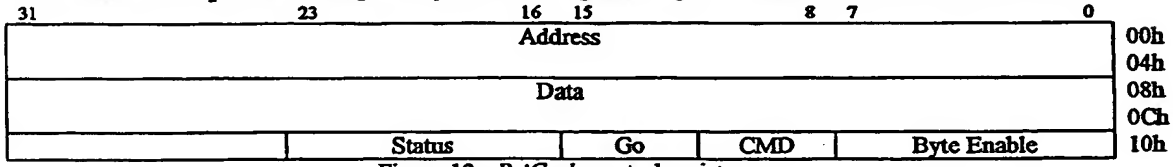


Figure 12 - *PciCycle* control register

Address field to be driven on PCI bus during the address phase.

Data field contains data to be driven to PCI bus during data phase of the write cycles or is a target for a data read in read cycles.

Byte Enable filed contains value to be driven on BE# lines during Byte Enable phase

CMD field contains command to be driven on PCI bus on C/BE# lines
When Bit0 of the GO field is set, it indicates that Address, Data, BE# and CMD are ready and cycle should be driven to PCI bus. When Bit1 of the GO field is set, MT101 should terminate cycle currently outstanding on the bus (de-assert ***FRAME#***). Bits 2-7 of GO field are reserved, should always be written as zeros and returned as zeros upon read.
HW clears updates Status field to reflect information about cycles' completion status as shown in Table 2.

Encode	Status
'1xx	Cycle in progress
'000	Normal completion of the cycle
'001	Re-try/disconnect
'010	Master-abort
'011	Target-abort

Table 16 - PCI cycles completion status

All fields of *PciCycle* register are cleared (zero) at HW reset

PCI compatibility – ordering rules

This section summarizes PCI ordering rules and how they are implemented in MT101/102 system.
Following are basic assumptions behind implementing the PCI ordering rules, that pose requirements to configuration SW:

1. NGIO fabric never reorders cells within same channel
2. All reordering is done on PCI bus or within PCI target or PCI master units of MT101/102
3. The NGIO fabric buffers (receive queues, transmit queues) are used strictly as 'shock absorbers'. They are never used to extend queues in PCI master or target unit. DR Capacity field in configuration headers should be used to allocate the Delayed Request queue in PCI master between the channels to implement this rule.

Rules 1,2,3,4 – Noone can pass previously accepted posted memory write

Hooks designed to implement this rule:
This rule is preserved strictly within the same channel – PCI slave will always issue NGIO cells in order PCI cycles were accepted. No re-ordering within same priority (channel) will occur within NGIO fabric. PCI master will always issue cycles on PCI bus in order they were received from NGIO fabric. Once posted write cycle issued on PCI bus, no other cycle will be issued until posted memory write is completed, and no read response cells are allowed to pass to the PCI target unit to prevent read completion passing previous write.

Rule 5 – A Posted Memory Write must be allowed to pass delayed requests

The enforcement of this rule is done through Delayed Request buffer on PCI master and its allocation to different channels in the network. The number of delayed requests to the same PCI bus cannot exceed capacity of its Delayed Requests buffer. Thus, all packets that are targeted to this PCI bus will arrive to the Delayed Requests buffer. Posted memory write does not consume an entry in Delayed Requests buffer, and it will be able to pass all previously-issued delayed requests stored in the PCI Master Delayed Requests buffer.

Rule 6 – Delayed Write Completion must be allowed to pass delayed requests

All Delayed Requests that cannot be completed will be stored in Delayed Request buffer in the PCI Master unit. Delayed Write Completion will pass them while arriving to the PCI target unit.

Rule 7 – A Posted Memory Write must be allowed to pass Delayed Completion

Posted memory write will pass Delayed Completion in the 'fork' between PCI Master and PCI Target. Delayed Completion cells will always be accepted by PCI Target (buffer is allocated while issuing the request packet).

PCI compatibility – bus cycles support

MT101/102 supports all PCI bus features and cycles except for:

1. LOCK# signal and functionality is not supported
2. Special cycles on the PCI bus are ignored

3. Type1 to special cycles conversion is not supported

SW generation of NGIO cells

MT101 provides a capability to generate and accept NGIO cell from NGIO fabric. This is capability is provided through System NGIO Port, which contains two 292-byte data structures and control register that are accessible as MT101 internal registers. The first structure – *OutBoundCell* – can be written by SW as a valid NGIO cell. After data is written to *OutBoundCell*, a *outbound_full* bit set in *SystemPortDoorbell* register, which initiates a send process. HW calculates the CRC for the cell (appends 32 bits) while sending the cell to the fabric. After cell has been sent to the NGIO fabric, HW clears *outbound_full* bit in *SystemPortDoorbell* register, indicating that *OutBoundCell* is empty, and new cell can be filled in. The second 292-bit data structure – *InBoundCell* – is used to accept new cells from the fabric. Cells targeted to System Port are stored in *InBoundCell* and *inbound_full* bit is set in *SystemPortDoorbell* register, indicating that new cell arrived. SW reads the data structure and clears the *inbound_full* bit, enabling new cell to arrive. As long as *inbound_full* bit is set, new cell that arrives to System Port will be dropped, and *dropped_cell* counter will be incremented, so SW can have a notice of dropped cells. Figure 13 illustrates System Port configuration registers.

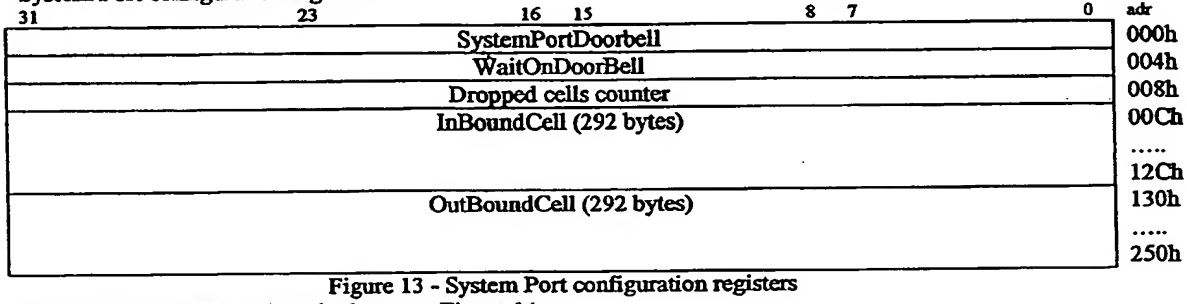


Figure 13 - System Port configuration registers

SystemPortDoorbell register is shown on Figure 14.

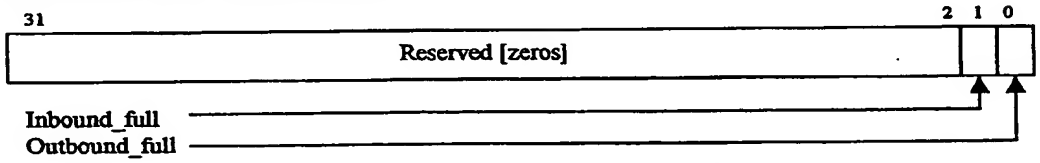


Figure 14 - SystemPortDoorbell register

WaitOnDoorBell register is used to wait till SystemPortDoorbell register is assigned in HW value that is written to WaitOnDoorBell register. On write to WaitOnDoorBell register, HW does not return acknowledge until value of SystemPortDoorbell register does not match value written to the WaitOnDoorBell, and no more configuration register access can start. This mechanism will enable to initiate entire NGIO fabric from S-EPROM. Note that careless use of this mechanism can hang the system, as access to all control registers can be blocked forever. This functionality is enabled only for accesses originated by S-EPROM.

SystemPortDoorbell, *WaitOnDoorBell* and *Dropped Cell Counter* fields of the System Port register are cleared (zero) at HW reset. Contents of *InBoundCell* and *OutBoundCell* fields is undefined.

NGIO boot

MT101/102 can be booted through NGIO boot mechanism, as specified in NGIO spec. Although MT101/102 does not implement HCA function to full extend, its architecture provides the capability to boot the entire system through NGIO interface. This can be done by generating NGIO cells explicitly through the system port mechanism of MT101.

NGIO cells arriving to FSA can alter control registers of the MT101/MT102. It will usually be Priority15 messages, although messages with other priorities arriving to the FSA will be treated as configuration messages as well.

Configuration messages are messages whose destination MAC matches FSA MAC address, and CMD field is FMPGet() or FMPSet(). The address of control register t be accessed and number of registers are defined in COD and COD_INDX fields, as explained in FMP access to configuration registers section. Configuration channels are non-connected, no acknowledge will be sent back to the request queue except implicit FMPGet() in response to FMPSet() message.
The data payload of the cell contains address of internal register to be accessed, command (read, write) and number of registers to be read. Configuration message can be either direct route or MAC-addressed. Data Payload format of the direct-routed NGIO configuration message is presented in Figure 15 and Data Payload format of the MAC-routed configuration message is presented in Figure 16.

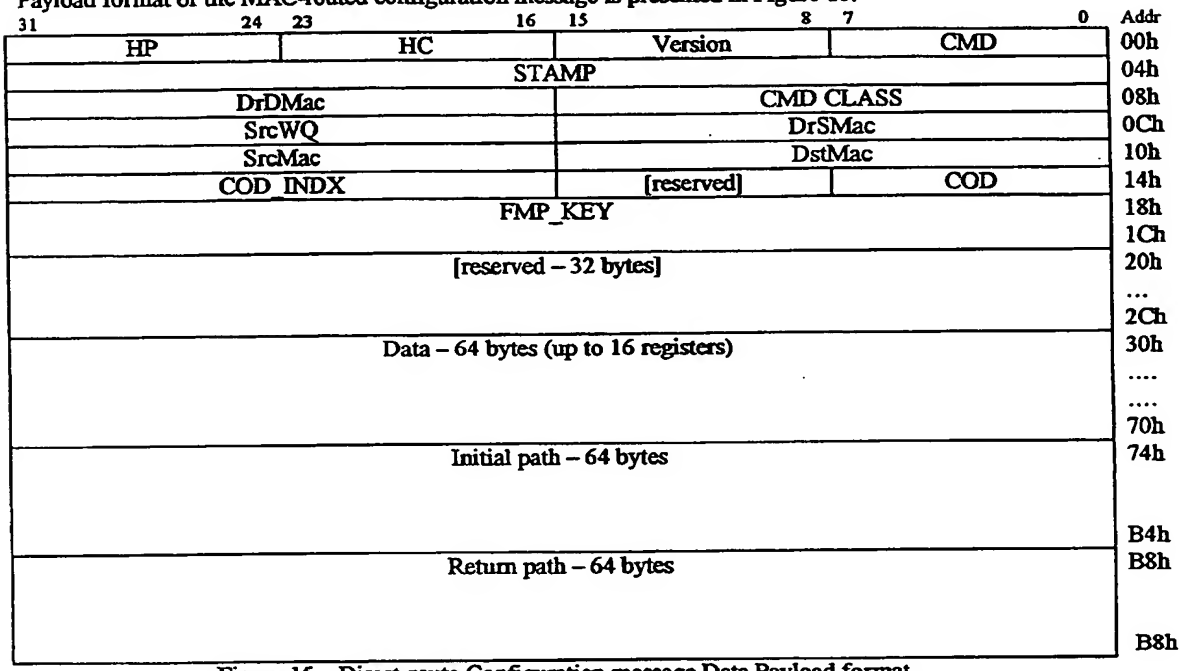


Figure 15 – Direct-route Configuration message Data Payload format

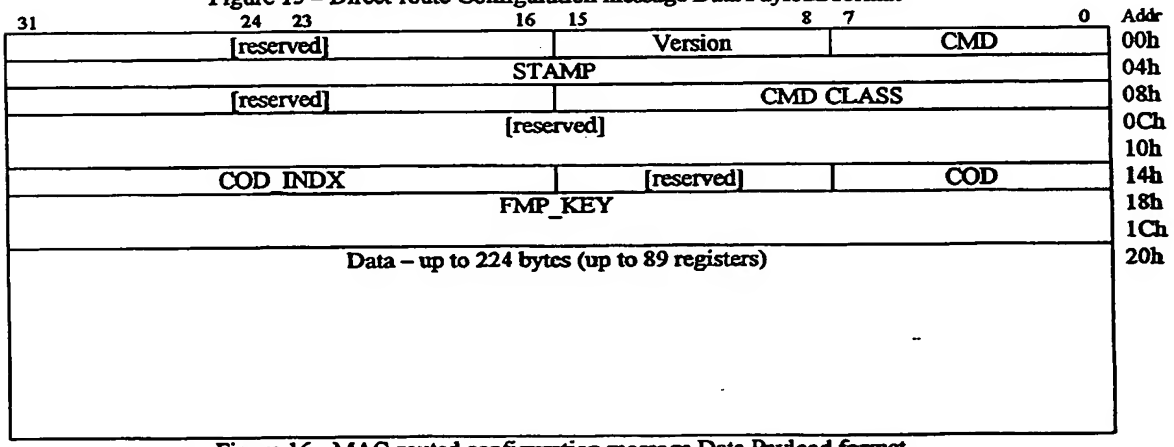


Figure 16 - MAC-routed configuration message Data Payload format

CMD field specifies whether it is CR read (FMPGet()) or CR Write (FMPSet()) operation.
Number of registers field specifies number of registers to be accessed. Note that with direct-routed message only 16 registers can be accessed by a single message.

MAC-routed FMPGet() message does not need to contain 224 bytes of data; the response message should append data as required by the number of registers accessed.

MT101/2 system initialization flow

MT101/102 system can be initialized from PCI, CPU or from a single S-EPROM, which is attached to MT101. In latter case, system configuration is embedded in S-EPROM programming (e.g. contents of S-EPROM at reset much be consistent with system configuration and topology.
MT101 enables to program S-EPROM, so once system configuration was changed, it can be re-programmed to update topology and system configuration information.. After re-programming system can again be boot off S-EPROM without PCI or CPU involvement.
Needless to say, that MT01/102 system can be initialized with various combinations of S-EPROM, PCI and CPU – as long as they agree with each other.

Events' generation and handling

MT101 and MT102 can generate and/or forward events to be delivered to Fabric Manager. This section specifies in details the mechanism of event's generation and delivery.

Event's generation

Once event is generated, it is delivered to fabric manager by FMP_TRP_REQ_MSG. If fabric manager interfaces with MT101/102 network through PCI or CPU endpoints, there are two options provided for message delivery:

1. MT101/102 end-point will write the incoming FMP cell to memory and ring the doorbell.
2. MT101/102 end-point will keep the arrived FMP cell in InBoundCell register of the SystemPort and ring the doorbell

In first case multiple FMP trap messages can arrive before the first one is handled by SW. It is SW responsibility to avoid FMP trap stack overflow (e.g. SW should poll the FMP traps stack). In second case only one (first) trap message will be kept until read by SW. In both cases HW interrupt can optionally be asserted upon new trap arrival (doorbell)

HW interface for event delivery

All events are delivered to SW through FMPs. FSA is exclusively responsible to deliver event to SW, implementing following steps:

1. Set appropriate bits in Cause Register
2. Construct FMP using EventFMPTemplate upon event request generated by HW.
3. Send this FMP to Fabric Manager
4. Wait for FMP acknowledging the event (clearing bits in Cause Register)
5. Re-send event FMP in case interrupt acknowledge FMP did not arrive within pre-defined time
6. Cease re-sends after ResendCount exceeded.

Figure 17 defines the Event FMP format. Shaded fields are taken from EventFMPTemplate register. Reserved fields are filled with '0.

Byte3	Byte2	Byte1	Byte0
Dest. WQPN (byte1)	Destination MAC		Version / Priority
Source WQPN (byte1)	Source MAC		Dest. WQPN (byte2)
[reserved]	PSN	Opcode	Source WQPN(byte2)
[reserved]	Cell payload length		CSN
HP	HC	Version	CMD
Stamp			
[reserved]		CMD Class	
CauseRegister			

Figure 17 - event FMP format

Control and status summary – errors and performance monitoring

Link errors are handled through Error Headers depicted in figures below. The definition is derived from NGIO Performance Management concept. Refer to Switch Spec, chapter 6 for more details and explanations,

PCI Performance Management Header

PCI port can encounter additional errors due to following reasons:

1. BAR/Limit range mismatch – overlap in channels' address space
2. IDSEL/secondary bus mismatch – device number is not covered by IDSEL mask (master abort)
3. IDSEL/secondary bus mismatch – device number is covered by more than one channel
4. Type1/0 conflict – wrong secondary/subordinate bus programming
5. Other errors that I could not think about now ☺

These errors are due to bugs in configuration SW. If such an error encountered, the respective cell is dropped and error occurrence is logged in PCI Error Counter.

31 24	23 16	15 8	7 0	Addr
Illegal response [target]				00h
Illegal response Limit [target]				04h
PCI Error counter [target]				08h
PCI Error Limit [target]				0Ch
PCI Error address [target]				10h
				14h
NACK posted write counter [target]				18h
Posted writes NACK Limit [target]				1Ch
NACK non-posted write counter [target]				20h
Non-posted writes NACK Limit [target]				24h
NACK reads counter [target]				28h
NAC reads Limit [target]				2Ch
Sequence Error counter [target]				30h
Sequence Error limit [target]				34h
Read MemLifeTime [target]				38h
Read reply timeout counter [target]				3Ch
Read reply timeout Limit [target]				40h
Non-posted Write MemLifeTime [target]				44h
Non-posted Write reply timeout counter [target]				48h
Non-posted Write reply timeout Limit [target]				4Ch
Posted Write MemLifeTime [target]				50h
Posted Write reply timeout counter [target]				54h
Posted Write reply timeout Limit [target]				58h
Reserved			Channel Hdr # [target]	5Ch
Illegal request [master]				60h
Illegal request Limit [master]				64h
PCI Error counter [master]				68h
PCI Error Limit [master]				6Ch
PCI Error address [master]				70h
				74h
Unable to complete Write counter [master]				78h
Unable to complete Writes Limit [master]				7Ch
Unable to complete read counter [master]				80h
Unable to complete read Limit [master]				84h
Sequence Error counter [master]				88h
Sequence Error limit [master]				8Ch
DR Capacity Exceeded counter				90h
DR Capacity Exceeded Limit				94h
PCI retry timeout [master]				98h
Reserved			Channel Hdr # [master]	9Ch
PCI error cause				A0h
PCI error mask				A4h

Figure 18 - PCI Performance Management Header

All fields of the PCI Performance Management Header are cleared (zero) at HW reset.

Illegal response counts number of illegal Response packets arrived to PCI. This includes illegal opcode (SEND response), channel mismatch, etc.

PCI error counters count number of PCI errors reported (e.g. parity error, configuration error etc.). The address of the cycle that resulted in error is stored in PCI Error address field.

PCI Error Address contains the address used in the PCI cycle that first caused error. Write to this register is enabled only after it has been read or PCI Error Counter is zero.

NACK counters (posted write, non-posted write, read) count number of NACKs received for the respective request.
Sequence Error counter counts number of sequence errors (as defined in NGIO spec)
Reply timeout counters (read, write) count number of timeouts occurred for the respective request. Write counter serves both posted and non-posted writes.
DR Capacity Exceeded counter counts number of times that number of pending delayed requests on any PCI channel is more than specified in DR Cap field of the PCI Master Channel Header.
Channel Header field contains the number of channel header that last encountered an error (data error, sequence error, NACK, timeout on reads etc.)
Error counters and limits are cleared at reset. Each time limit register is updated (through.CR write operation), counter is loaded with same value. Each time event occurs, counter is decrement. On transition from '1 to '0 value of the counter, the respective bit is set in PCI Cause register. Event is generated if is enabled by the PCI Error Mask (respective bit is not cleared in PCI Error Mask register).
PCI Error Cause Register is defined in Figure 19

Bit	Cause
0	Illegal response limit exceeded
1	PCI error limit exceeded [target]
2	Posted writes NACK limit exceeded
3	Non-posted writes NACK limit exceeded
4	Read NACK limit exceeded
5	Target sequence error limit exceeded
6	Read reply timeout limit exceeded
7	Write reply timeout limit exceeded
8-15	RESERVED
16	Illegal request limit exceeded
17	PCI error limit exceeded [master]
18	Unable to complete write limit exceeded
19	Unable to complete read limit exceeded
20	Sequence error limit exceeded [master]
21	DR Capacity Exceeded limit exceeded
22-31	

Figure 19 - PCI Error Cause register

NGIO port performance management header

The NGIO port performance management and error reporting is defined in compliance to Performance Monitoring definition (Switch, section 6). Figure 20 defines the NGIO Performance Management Header

31 24	23 16	15 8	7 0	
PortTxOctets				00h
PortRxOctets				04h
PortTxCells				08h
PortRxCells				0Ch
PortRxErrors				10h
PortRxCellDiscards				14h
PortTxCellDiscards				18h
PortRxCellsTooShortErr				1Ch
PortRxCellsTooLongErr				20h
PortRxCellsCRCErr				24h
PortRxCellsDisparityErr				28h
PortRxCellsEncodeErr				2Ch
PortRxPriError				30h
PortRxDestRxPort				34h
PortTxLifetimeErr				38h
PortTxExcessFCErr				3Ch
PortTxActiveErr				40h
PortTxOctets Limit				44h
PortRxOctets Limit				48h
PortTxCells Limit				4Ch
PortRxCells Limit				50h
PortRxErrors Limit				54h
PortRxCellDiscards Limit				58h
PortTxCellDiscards Limit				5Ch
Internal Error counter				60h
Internal Error Limit				64h
NGIO Port Error Cause				68h
NGIO Port Error Mask				6Ch

Figure 20 - NGIO Port Performance Management Header

All fields of NGIO Port Performance Management Header are cleared (zero) at HW reset.
Italics fields defined in Switch Spec, section 6.
Internal Error counter counts error generated inside the MT101, as described in Internal data integrity section.
Port Error Cause register logs events by setting appropriate bit and event is generated if not masked by respective bit in the Event Mask register

Bit	Cause
0	PortRxOctets
1	PortTxCells
2	PortRxCells
3	PortRxErrors
4	PortRxErrors
5	PortRxCellDiscards
6	PortTxCellDiscards
7	PortTxLifetimeErr
8	PortTxExcessFCErr
9	PortTxActiveErr
10	Internal Error
7-31	reserved

Figure 21 - NGIO port error cause register

FSA performance management header – event generation side

FSA consolidates all events generated in the MT101, constructs a combined Cause Register, constructs FMP event message and sends it to the Active Fabric Manager MAC. Figure 22 defines the FSA Performance Header – the event generation part

31 – 24	23 – 16	15 – 8	7 – 0	
Consolidated Cause Register				00h
Event Mask				04h
Event Response timeout counter				08h
Event Response timeout limit				0Ch
Event Retry counter				10h
Event Retry limit				14h
EventFMPTemplate Register (Figure 23) [12 registers]				18h ... 44h

Figure 22 – FSA Performance Management Header, event generation part

All fields of FSA Performance Management Header are cleared (zero) at HW reset.

Consolidated Cause Register includes information about all events occurred in this device. Event's generation (FMP message) is generated only if corresponding bit in Event Mask register is set.

Bit	Cause	Bit	Cause
0	NGIO link down	16	Trap RDMA-Write timeout/NACK
1	NGIO link up	17	
2	NGIO RxQ err limit exceeded	18	
3	NGIO TxQ err limit exceeded	19	
4		20	
5		21	
6		22	
7		23	
8	PCI sequence error	24	NGIO Octets/Cells limit (either)
9	PCI RD/non-post WR bad response	25	
10	PCI posted WR bad response	26	
11	PCI interrupt INT	27	
12	PCI bus error	28	
13		29	
14		30	
15		31	

Table 17 - Consolidated Cause Register

EventFMPTemplate register is defined in Figure 23.

31 – 24	23 – 16	15 – 8	7 – 0	
Dest. WQPN (byte1)	*Destination MAC*		Version | Priority	00h
Source WQPN (byte1)	*Source MAC*		*Dest. WQPN (byte2)*	04h
[reserved]	PSN	Opcode	Source WQPN(byte2)	08h
[reserved]	Cell payload length		CSN	0Ch
HP [0]	HC [0]	Version	CMD	10h
Stamp				14h
DrDMAC		CMD Class		18h
SrCWQ		DrSMac		1Ch
SrcMac		*DstMac*	COD	20h
COD_IDX		[reserved]		24h
FMP_KEY				28h 2Ch

Figure 23 - EventFMPTemplate register

Fields in italic are alias to respective fields in the MT101 Global Configuration Header (Figure 25) and their reset values must match those of the Global Header. Table 18 specifies reset values of remaining fields.

Field	Value	C mment
Priority	FFh	FMP, priority 15
Version	1h	Version 1 (per link spec)
Source WQPN	0	
Opcode	4h	NGIO-send,
PSN	0	
CSN	0	
Cell payload len	0	
CMD	4h	Trap request message
Version	0	
HC	0	
HP	0	
Stamp	0	
CMD Class	0	
DrDMAC	0	
DrSMAC	0	
SrCWQ	0	
COD	0	
COD_INDX	0	
FMP_KEY	0	

Table 18 - EventFMPTemplate register reset values

FSA performance management header – event recipient side

Once FMP is generated, it is forwarded to Active FM MAC address. In MT101 systems FSA of the MT101 can serve as a destination of the Event Message. It provides basic HW hooks for SW interface – stores recipient message in internal register, can optionally translate it to RDMA-write packet and forward it to port with memory (e.g. PCI or 8-bit CPU). It also can optionally assert INT or SERR output of the MT101.

31 24	23 16	15 8	7 0	
Received Cause Register				00h
INT Mask				04h
SERR Mask				08h
RDMA-Write Mask				0Ch
Memory Stack stride				10h
Reserved			RDMA-Write priority	14h
RDMA-Write destination WQPN		RDMA-Write Destination MAC		18h
RDMA-Write source WQPN		*RDMA-write source MAC*		1Ch
Reserved		CSN	PSN	20h
RDMA-Write VA/MH				24h
				28h
RDMA-Write MH				2Ch
RDMA-Write Response timeout counter				30h
RDMA-Write Response timeout limit				34h
RDMA-Write Retry counter				38h
RDMA-Write Retry limit				3Ch
FMPTrap Door Bell register				40h

Figure 24 – FSA Performance Management Header – recipient side

All fields of FSA Performance Management Header are cleared (zero) at HW reset.

Upon FMPTrap() message arrival, FSA checks whether it is a destination for this message by examining destination MAC address. If destination MAC address matches its own address, FSA extracts Cause Register from the FMPTrap() message and stores it in the FSA Performance Management Header. If interrupt or SERR is enabled by the respective mask in the FSA Performance Header, FSA asserts INT or SERR pins.

If Memory Event Stack is masked, the arrived cell is stored in the InBoundCell register of the SystemPort, and SystemPort Doorbell is rung.

If Memory Event Stack is enabled by respective bit in the RDMA-Write Mask, FSA generates RDMA-Write message with MAC header fields specified in the FSA Perf rmance Management Header (09h-0Bh).
The data payload of RDMA-Write should contain MAC header of the original FMPTrap() message and cause register. The address pointer (RDMA-Write VA register) should be incremented by the value stored in Memory Stack Stride (sign-extended), so it will be ready for the next message.
If RDMA-Write was not acknowledged within RDMA-Write timeout limit after RDMA-Write retries limit or it was NACK'ed, FSA sets Trap-RDMA-Write timeout/NACK bit in the Consolidated Cause Register of the FSA Performance Management Header (event generation part). This may result is sending the FMPTrap() message to the destination as specified in the EventFMPTemplate of this device.
In order to avoid endless loops, RDMA-Write should be masked for Trap-RDMA-Write timeout event if destination of the FMPTrap() is the same FSA that issued the RDMA-Write.
Normally, Trap RDMA-Write messages should be sent with priority15 (FMPs) to non-connected destination, and acknowledge would be clearing Cause Register by SW. However, it is possible to send this message to connected/acknowledged channel (that should be configured ahead on the destination side).

MT101 Configuration registers Summary

This section summarizes configuration registers of MT101.

Global MT101 configuration (FSA)

Global MT101 configuration information which defines operation of entire MT101 device. Figure 25 defines Global MT101 configuration header. This header resides in FSA and managed by FSA. Different fields of the register can be altered (or not altered) by FMPs, refer to NGIO spec – FSI section.

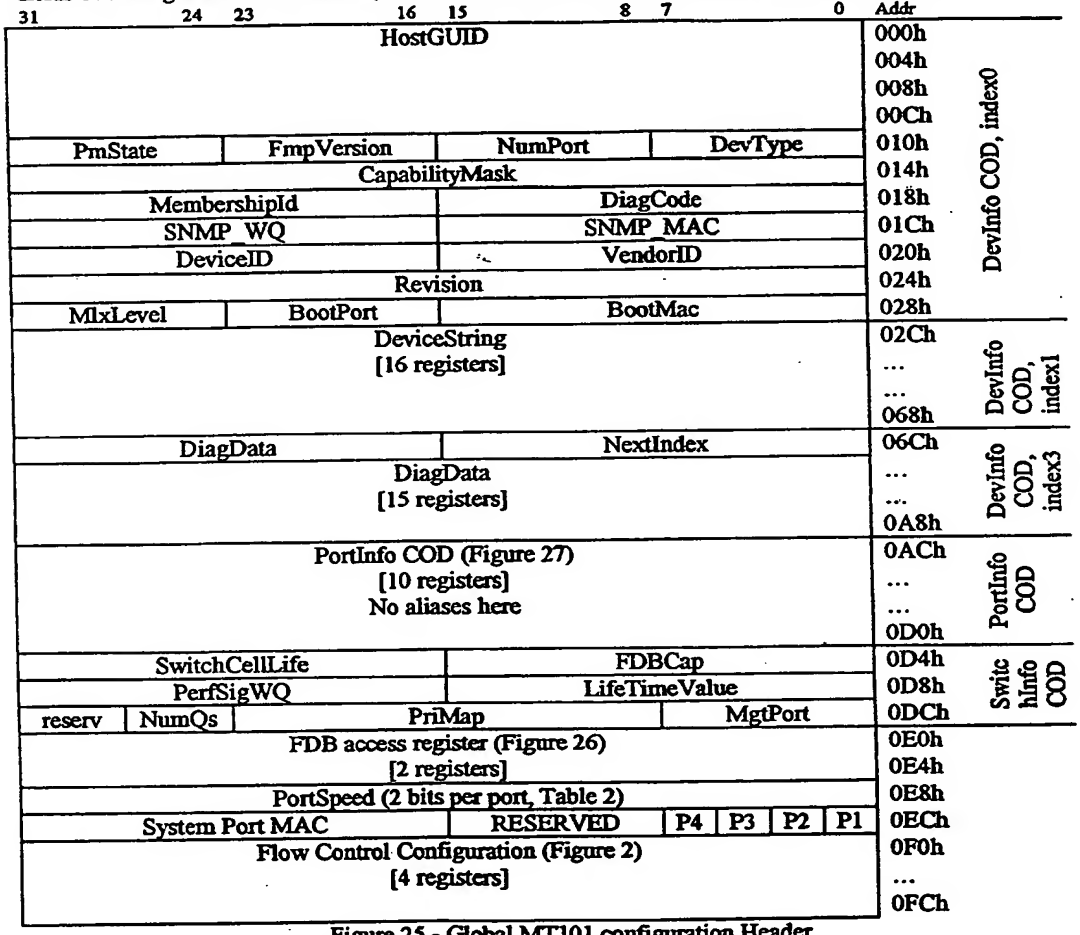


Figure 25 - Global MT101 configuration Header

FDB is being accessed through MT101 configuration space access in 4-entries chunks through two dedicated registers – FDBData and FDBArdCtrl. These registers are shown on Figure 26.

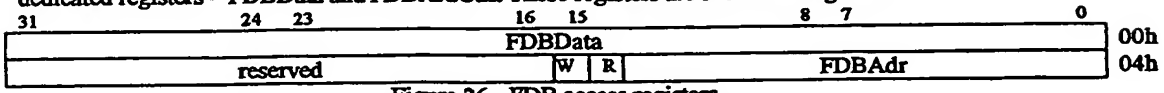


Figure 26 - FDB access registers

FDBAdrCtrl register contains 3 fields – FDBAdr filed (bits 13:0) define the 4-byte entry address (covering 4 FDB entries, compatible to FDB COD format and two control fileds – R (bit14) and W (bit15). If FDBAdrCtrl register is loaded with R field set, the FDB entry defined by FDBAdr field is read from FDB and placed to FDBData register. The configuration cycle is acknowledged only after read operation is

completed, and R field is cleared by HW. If FDBAdrCtrl register is loaded with W field set, the FDB entry addressed by FDBAdr field is loaded with content of FDBData register. The configuration cycle is acknowledged only after write operation is completed and W field is cleared by HW. Result of loading FDBAdrCtrl register with both W and R fields is undefined.
PortSpeed register defines transmit queue speed of each port. Bits 0:1 correspond to port0, bits 2:3 – port1, etc.
P4-P1 fields (bits 7:0) specify values (Table 4). Each field is 2-bit wide

Field	Value	Comment
HostGUID		Needs definition
DevType		Needs definition
NumPort	Ah	10 ports overall
FmpVersion	1	
PmState	0	
CapabilityMask	0	Nothing exotic supported
DiagCode		
MembershipId		
SNMP_MAC	0	
SNMP_WQ	0	
VendorID		Needs definition
DeviceID		Needs definition
Revision	0	
BootMac	0	
BootPort	0	
MlxLevel	0	No MLX supported
DeviceString		Need to write something funny
NextIndex	0	Single diag data (if at all ☺
DiagData	0	
FmKey	0	
ActiveFm	0	
TimeOut	0	No timeouts
ProtBit	0	No FM_KEY protection
FDBCap	4000h	16K FDB entries
SwitchCellLife	0	No timeout
LifeTimeValue	0	No timeout
MgtPort	0	
PriMap	FA50h	Seems to be inconsistency in the NGIO definitions, may need more bits
NumQs	4	4 priority queues suported
FDBData	0	
FDBAdrCtrl	0	
PortSpeed	FFFFh	All TxQs are set as very fast; RxQs as slow, which implies full buffering by default.
Buffering	FFFFh	Full buffering by default – always
Config	0	8 P2P bridges

Table 19 - Global MT101 Configuration Header reset values

NGIO port configuration

Figure 27 shows general template for PortInfo COD register, that will be instantiated in each NGIO port – native links, FSA and PCI. Fields in italic are common in all ports and implemented in FSA port only.

31 24	23 16	15 8	7 0	Addr
DevGuid [alias to HostGUID] *[implemented in FSA only, not implemented in NGIO ports]*				00h 04h 08h 0Ch
FmKEY *[implemented in FSA only]*				10h 14h
ActiveFM [FSA only]		*MacAddress[7] [FSA only]*		18h
TimeOut		ChanSigWQ		1Ch
PortStat | Fmnum	LinkSpeedSet	LinkSpeedSupport	LocalPortNum	20h
LoopBkEn (bit7), IsFSA (bit6), IsExternal (bit5), *Protection bit (bit4)*			rsrv	24h

Figure 27 - PortInfo COD remplate register

Figure 28 illustrates NGIO port configuration Header. These registers exist in every 'native' NGIO port (e.g. excluding ports that serve FSA, PCI etc.). Fields specified in italic are alias to respective filed in the Global Configuration Header (Figure 25)

31 24	23 16	15 8	7 0	Addr
Port performance management header (Figure 20) [28 registers]				00h 6Ch
				70h 94h
LiveLock, Prio3	LiveLock, Prio2	LiveLock, Prio1	LiveLock, Prio0	98h
reserved			TxQT | RS	9Ch
Flow Control Configuration (Figure 2) [4 registers]				A0h A4h A8h ACh

Figure 28 - NGIO port configuration Header

RS field (bits 1:0 at offset 9Ch) define the receive qucue speed (Table 2)
TxQT field (bits 7:2 at offset 9Ch) define number of data chunks to bc buffered in the transmit queue before transmit starts on the link

Field	Value	Comment
MacAddress	0	
ChanSigWQ	0	
LocalPortNum	0-9	Loaded according to its placement
LinkSpeedSupport	0	2.5Gb/sec
Fmmum	0	
PortStat	1	Initializing
IsExternal	x	0 for FSA and PCI ports '1 for the rest
LoopBkEn	0	Disabled
LiveLock 0,1,2,3	0	LiveLock disabled by default
RxQ spd	0	Slow receive queue

Table 20 - NGIO Port Configuration Header reset values

PCI configuration

Figure 30 presents summary of PCI Configuration Header

[7] MacAddress is alias in all NGIO ports to FSA. PCI port has it own distinct MAC value.

31 24	23 16	15 8	7 0	Addr
PCI Device function configuration Header – Functions 0 to 7 (PCI spec) [8 functions, 16 registers each]				000h 1FCh
PCI Target Channel Header (Figure 6) – channels 0 to 31 [32 channels, 8 registers each]				200h 5FCh
[reserved] – in case more channels needed				600h 9FCh
PCI Master Channel Header (Figure 8) – channels 0 to 31 [32 channels, 2 registers each]				A00h AFCh
[reserved] – in case more channels needed				B00h BFCh
P2P Port configuration registers (Figure 10) – port 0 to 7 [8 configuration registers, 4 registers each]				C00h C7Ch
PciCycle header (Figure 12) [5 registers]				C80h C90h
PCI Performance management header (Figure 18) [42 registers]				C94h D38h
TargetWqpBase		MasterWqpBase		D3Ch
PCI port MAC address		Config (Table 8)	DR Cap	D40h
[reserved]				D44h DFCh
PCI NGIO port Configuration Header (Figure 28) [44 registers]				E00h EB0h

Figure 30 - PCI Configuration Header

WQP base registers and PCI MAC address are cleared (zero) at reset.
DR Cap field specifies number of Delayed Request buffers implemented in the PCI Master unit. It is set to 64 at reset.

Miscellaneous configuration registers

Figure 32 shows miscellaneous configuration registers of MT101

31 24	23 16	15 8	7 0	Addr
SoftReset (Table 7)				00h
EPROM control (Figure 3) [2 registers]				04h 08h
Timer divider (to make 32micro-sec clock out of system clock)				0Ch
Timer (Figure 5) [2 registers]				10h 14h
Clock shutdown (Table 22)				18h
SERDES Configuration register (Figure 4) [3 registers]				1Ch 24h

Figure 32 - Miscellaneous configuration registers

Field	Value	Comment
Timer Divider	FA0h	Will make 32micro-sec assuming internal clock is 125Mhz
Clock shutdown	0	No clocks are closed

Table 21 - Miscellaneous configuration registers reset values

Bit	Unit
0	PCI
1	CPU interface
2	S-EPROM interface
3	FSA
4	NGIO port0
5	NGIO port1
6	NGIO port2
7	NGIO port3

Bit	Unit
8	NGIO port4
9	NGIO port5
10	NGIO port6
11	NGIO port7
12	NGIO port8
13-32	RESERVED

Table 22 - Clock Shutdown register

Clock shutdown register is defined in Table 22. If bit is set, the clock to respective unit is disabled (blocked). Used to reduce power dissipation in the units that are not functional.

Configuration space summary

Figure 34 shows configuration space of MT101, including address assignments of the configuration registers.

PCI Configuration Header (Figure 30) [??? registers + reserved]	0000h 0EB0h
[RESERVED] for future PCI expansion ☺	0E50h 0FFCh
NGIO port configuration – port 0 to 7 (Figure 28) [8 ports, 64 possible registers each, 44 used now]	1000h 17FCh
RESERVED For future NGIO ports expansion ☺	1800h 1FFCh
System Port configuration (Figure 13) [149 registers]	2000h 2250h
FSA Performance Management Header, event generation part (Figure 22) [18 registers]	2254h 2298h
FSA Performance Management Header – recipient side (Figure 24) [17 registers]	229Ch 22DCh
MT101 Global Configuration registers (Figure 25) [64 Registers]	22E0h 23DCh
[RESERVED] For future global expansions ☺	23E0h 2FFCh
Miscellaneous configuration registers (Figure 32) [10 Registers]	3000h 3024h

Figure 34 - MT101 configuration space summary

MT101 configuration registers access

MT101 configuration registers can be accessed from PCI, from S-EPROM, from 8-bit CPU and from FSA. All registers are treated as 32-bit values. Reading register with reserved field will return zero on reserved field; writing value other than zero to reserved field will result in undefined behavior.
Restrictions apply on accessing certain registers from PCI of from Fabric Manager – as defined in NGIO spec or PCI spec respectively. FMPs restriction applied only for NGIO-specified CODs. There are no restrictions accessing configuration registers from S-EPROM port, 8-bit CPU port or from FMP that uses MT101-specific CODss, although FM_KEY protection still holds to avoid security hole.

PCI access to configuration registers

Registers are mapped to PCI address space. Each register occupies 4 bytes and they can be accessed as DWORD entities only. BAR register of function0 PCI header holds the memory tag for the configuration registers. PCI definition restrictions apply to accessing PCI configuration headers.

EPROM access to configuration registers

EPROM interface unit reads contents of S-EPROM and loads configuration space as described in the Serial EPROM – initialization section. All registers can be written from EPROM interface

CPU access to configuration registers

CPU can only access internal registers of MT101. If MT101 is selected (CS# asserted) the 12 bits of the memory address that are connected to the MT101 are interpreted as control register address, and access (read or write) is performed. All registers can be read or written from CPU interface.

FMP access to configuration registers

Control registers can be accessed through FMP message (FMPGet() or FMPSet()). MT101 supports following NGIO standard CODs for FMPGet() and FMPSet():

1. Devinfo COD – indexes 0, 1 and 3
2. PortInfo COD – indexes from 0 to 9 (10 ports – 8 NGIO, 1 FSA and 1 PCI)
3. SwitchInfo COD
4. FDBEntry COD – indexes 0 through 255 (support 16K entires)

MT101 DOES NOT support in HW FMInfo COD, Notice COD, TCAINfor COD, TcaErroinfo COD, Inforninfo COD, MlxInfo COD, Failover COD. FMPSet() with these CODs will have no effect on MT101 operation, and FMPGet() with these CODs will return zero.

MT101 defines two additional CODs to access configuration registers.

MT101 register access COD, CODID: 64

Used to access small number of registers (one to 16) in a single FMP packet. The COD_INDX field defines the number of registers to be accessed and address of the first register in the pack. Four MSBs of the COD_INDX specify number of registers to be accessed (zero value corresponds to single register access, FFh value corresponds to 16-register access). Remaining 12 bits specify the address of the first register to be accessed.

MT101 bulk register access COD, CODID: 65

Used to access larger number of registers (one to 64) in a single FMP packet. The COD_INDX field defnies number of registers to be accessed and address of the first register in the pack. Six MSBs of the COD_INDX specify number of registers to be accessed (zero value corresponds to single register access, 3FFh value corresponds to 64-register access). Remaining 10 bits are used to form the address of the first register in the pack by shifting it left two places. In other words, accesses to more than 16 registers in a single FMP packet must be aligned to quad register address.

MT101 signal description

PCI interface

MT101 implements 64-bit PCI interface, operating up to 66Mhz

Embedded CPU interface

MT101 has a glue-less interface to MPC860 Motorola processor. MPC860 must be configured to 8-bit port size to work with MT101. While accessing MT101, the CPU can only address 32-bit register, and all accesses should be implemented as 4-beat bursts. MT101 monitors 12 bits of the CPU address, TS#, CS# and R/W# signals. It drives/samples 8 bits of the data bus (that should be connected to lowest byte of the CPU data bus) and TA# signal.

S-EPROM interface

MT101 has a glue-less interface to MicroWire serial EPROM, MicroChip 93C76/86 series. The data sheet is available in Outlook public folder

JTAG interface

MT101 implements IEEE-compatible JTAG test port.

MT101 external signals summary

Name	Description	I,O,I/O	Voltage	#
	PCI interface			
AD[63:0]	PCI address/data	I/O	3.3VD	64
C/BE[7:0]#	PCI command/byte enable	I/O	3.3VD	8
PAR	PCI parity	I/O	3.3VD	1
PAR64, REQ64#, ACK64#	PCI 64-bit support	I/O	3.3VD	3
FRAME#	PCI interface control	I/O	3.3VD	1
TRDY#	PCI interface control	I/O	3.3VD	1
IRDY#	PCI interface control	I/O	3.3VD	1
STOP#	PCI interface control	I/O	3.3VD	1
DEVSEL#	PCI interface control	I/O	3.3VD	1
IDSEL#	PCI interface control	I/O	3.3VD	1
PERR#, SERR#	PCI error report	I/O	3.3VD	2
REQ#	PCI arbitration	O	3.3VD	1
GNT#	PCI arbitration	I	3.3VT	1
SBO#, SDONE#	PCI cache support	I/O	3.3VD	2
PCLK	PCI clock	I	3.3VT	1
	JTAG			
TDI, TCK, TMS, TRST#	JTAG, IEEE 1149.1	I	3.3VT	4
TDO	JTAG, IEEE 1149.1	O	3.3VD	1
	NGIO ports			
NP[7:0]DI[9:0]	NGIO Data In	I		80
NP[7:0]DO[9:0]	NGIO Data Out	O		80
NP[7:0]CLKI	NGIO port clock input	I		8
NP[7:0]CLKO	NGIO port clock output	O		8
NP[7:0]VREFI	NGIO port reference voltage input	I		8
NP[7:0]VREFO	NGIO port reference voltage output	O		8

Name	Description	I,O,I/O	Voltage	#
SERDES configuration interface (assuming AANetCom SERDES)				
DVAD[4:0]	Device Address	O		5
MDIO	Management Data I/O	I/O		1
MDC	Management Data Clock	O		1
Serial EPROM interface (see MicroChip 93C76/86 for details)				
SROMADR[3:0]	S-EPROM address – upper 4 bits	O	3.3VD	4
SROMCLK	S-EPROM clock	O	5VT	1
SROMDI	S-EPROM Data In	I	5VT	1
SROMDO	S-EPROM Data Out	O	3.3VD	1
System monitoring/scan				
SDO	Scan data out	O		1
SCLK	Scan clock out	O		1
SSTRB	Scan Strobe (indicates first bit in the chain)	O		1
CPU interface (see MPC860 spec for details)				
C_ADR[11:0]	Address, used to access control register	I		12
C_TS#	Transfer Start	I		1
C_CS#	Chip select, qualifies CADR and TS#	I		1
C_DAT[7:0]	Data bus	I/O		8
C_TA#	Transfer acknowledge (like RDY)	O		1
C_CLK	CPU interface clock	I		1
C-RW#	Read/Write#	I		1
Miscellaneous global signals				
RST#	RESET	I	3.3VT	1
INTI	Interrupt In	I	3.3VT	1
INTO	Interrupt Out	O	3.3VD	1
CCLK	Core clock	I		1
TOTAL				332

Table 23 - MT101 functional pins summary